The Design and Implementation of the Mansion Mobile Agent System

van 't Noordende, G.J.

2015

document version
Publisher's PDF, also known as Version of record

.

THE DESIGN AND IMPLEMENTATION OF THE MANSION MOBILE AGENT SYSTEM

.

VRIJE UNIVERSITEIT

THE DESIGN AND IMPLEMENTATION OF THE MANSION MOBILE AGENT SYSTEM

ACADEMISCH PROEFSCHRIFT

ter verkrijging van de graad Doctor aan
de Vrije Universiteit Amsterdam,
op gezag van de rector magnificus
prof.dr. F.A. van der Duyn Schouten,
in het openbaar te verdedigen
ten overstaan van de promotiecommissie
van de Faculteit der Exacte Wetenschappen
op dinsdag 3 februari 2015 om 13.45 uur
in het auditorium van de universiteit,
De Boelelaan 1105

door

Guido Johannes van ’t Noordende

geboren te De Cocksdorp, Texel

promotoren: prof.dr. A.S. Tanenbaum
prof.dr. F.M.T. Brazier

.

.

I have many to thank.
For being there, for support, for help, and for supervision.
You know who you are.
Thank you.

Ik heb velen te bedanken.
Voor er zijn, voor ondersteuning, voor hulp en voor supervisie.
Je weet wie je bent,
Bedankt.

Table of Contents

Chapter 1
Introduction

Many Internet-based distributed systems have been designed over the years. The World Wide Web (the Web) is a well-known example of a system that provides a way to navigate the Internet [24]. Besides the Web, other ways have been devised to locate distributed information. For example, FTP, bulletin board servers, and gopher (a distributed document retrieval protocol used mainly with text-based clients) [12] are alternative mechanisms to locate information on servers on the Internet, often using a directory-like structure. A problem with most of these systems is that a user has to know exactly on what server a file is in that system's file hierarchy. This way of operating is difficult and error prone.

The Web improved this by providing concepts for navigation. Documents contain hyperlinks to other documents that users can follow to go from one page to another. Because the Web is large and because it contains such a diversity of content, it remains difficult to organise and locate information. Hyperlinks are Universal Resource Locators (URLs) that can point to any other Web page. Although often a relation exists with the originating Web page, it is often unclear what information a hyperlink points to. The current-day solution to this problem is to construct search engines that index the Web so that it can be searched, but there are situations where this does not suffice.

The Web does not provide a solution for all applications. For example, an application may want to search images for a hidden feature that is difficult to find. Some features in an image (such as the existence of a boat in a picture) may be recognisable and can be indexed with relative ease using existing image processing tools, but rarer or more specific shapes may be harder to find. Finding a specific tumour growth within a three-dimensional brain scan may be completely infeasible except for very specialised image analysis software. The Web has another problem in this case: patient information can be privacy-sensitive and may not be made available on systems external to the system where the information was collected. It thus is not available to the Web, including to search engines. For intellectual property, such as music files, movies, or other digital content the same may hold. The question is: how to locate and process information that cannot be downloaded?

Mansion, the model presented in this thesis, provides an alternative to the Web. Mansion provides a collection of independent domain-specific worlds, each of which consists of hyperlinked rooms that have a predetermined structure. Active, autonomous processes called *agents* can travel these worlds to do their jobs.

Mansion presents a new programming paradigm for mobile agents, centered around structured, distributed worlds that contain hyperlinked rooms. A world provides a closed environment that has a predictable, application-specific hyperlink layout to allow agents to locate rooms with certain content; its hyperlink structure and other essential properties are determined by a world designer.

A world allows for organising information in an application-specific way, so that this information can be found and processed—even if it cannot be made available for indexing or downloading. Information can be provided in a world by placing it in a room. Agents visit rooms by following hyperlinks which are annotated to help agents find their way. When an agent follows a hyperlink, it physically migrates to the room where the data is, so it can inspect the data there—or meet other agents there. A mechanism called *confinement* allows room owners to control the export of information found by an agent, in case leakage of data must be prevented.

Agents can go to rooms to find other agents to negotiate deals or speed up their search. A group of agents may be sent out by companies to negotiate about the time and price at at which some product, such as electricity, will be delivered. Agents may collectively search for difficult to find information in high-resolution scans of art in museum rooms, using special algorithms. Agents may be looking for movies or audio fragments, search for travel advice, or they may be searching archives for scans of medieval poetry. Agents may also represent medical researchers that visit databases containing sensitive medical images or DNA data.

Throughout this thesis, we will explain the proposed model, how it can be implemented and what it can be used for. The above and other examples will be discussed. A prototype has been constructed and will be discussed at length in chapter 10.

1.1. Overview of the Mansion paradigm

Mansion presents a new paradigm to structure words for specific applications. It is based on the notion of closed, application-specific worlds. Each world consists of a potentially large set of *rooms* that contain content. Rooms are interconnected by (unidirectional) hyperlinks. Rooms contain hyperlinks, objects containing content, and agents. Agents interact with the world by following hyperlinks and by searching rooms, inspecting content there. Agents can also communicate with other agents. An agent can only be in one room at a time and can only access content in that room. In contrast to the Web, hyperlinks are part of a room and not embedded in documents.

A world provides an application-specific environment. This environment is distributed: content can be added by different users, typically on their own systems which they made part

of the world. All entities in a world (agents, content, and hyperlinks) are annotated using *attribute sets* that can be inspected by agents to facilitate search. The semantics of attributes are defined per application. Users submit agents into a world, which navigate through a set of rooms by following hyperlinks. An agent autonomously executes a task on behalf of its owner, either disconnected from its owner or interactively. Agents are *mobile:* they physically migrate to the location where the data is and access content there; agents are automatically migrated when following a hyperlink.

A world is physically distributed, with rooms added and managed by different people and organisations. A particular challenge is to maintain controllability over a world on the one hand—ensuring consistency and integrity of the world's layout and attribute sets, for example, to ensure the world remains searchable—yet avoiding that a world owner needs to or can control every detail of it. Like in the web, freedom of users to place their own content in a world and to search a world's content is considered imperative. A world designer has a large degree of control over the world's topology and its content, to ensure it is a coherent, application-specific environment, but should not be able to control what content is placed in it. Control over who may place what content in a world is decentralised and lies with the room owner.

The design presented in this thesis is a tradeoff between controllability and security considerations concerning world design and management on the one hand, and autonomy and privacy constraints of users of a world on the other hand. This can be formulated as a question driving the research in this thesis:

Can we facilitate construction of application-specific distributed worlds, in which users are free to add content, but where the system can be sufficiently structured so that it remains understandable by end-users and their agents, striking a balance between controllability and security of the system on the one hand and flexibility and autonomy of users to implement their algorithms to find rooms and content—and to process this content—on the other hand?

This thesis describes the motivation, the requirements, the architectural design, usage scenarios, and various aspects—such as security—that impacted the design and implementation of Mansion and the middleware system that supports it. This includes the tradeoff apparent from the above question.

The remainder of this chapter describes how Mansion relates to the Web, then describes other related work, and then enumerates the main contributions of this work.

1.2. Exploring Mansion: an alternative to the Web

This thesis describes a novel approach to constructing distributed systems. Mansion introduces the concept of a mobile agent in the context of a Web-like distributed environment. Mobile agents are mobile programs that can be programmed by their user and which can autonomously navigate through a world, detached from their user. Because agents are (mobile) programs, they can be tailored to a specific application.

Mansion makes it possible to define application-specific worlds that contain a set of rooms and world related content. For example, there may be a book world or an electricity auction world. The former allows agents to find books on certain topics in certain rooms; the latter may have rooms in which agents negotiate contracts or bid in an electricity market. Related rooms can have hyperlinks between them, possibly corresponding to some application-specific topological structure defined by a world owner. Hyperlinks and content are described using attributes that allow agents to find their way autonomously without user interaction. Alternatively, they can search a world with user interaction, as long as the data they access is not sensitive (otherwise, the agent's current room should disable communications with the outside world).

Because the structure of a Mansion world can be tailored to the needs of a specific application (particularly, how hyperlinks are annotated and organised), the idea is that agents can navigate their environment quickly and independently. By creating different worlds for different applications, world designers as well as users can shield their agents from other applications, unrelated content, and unrelated agents, making them operate more effectively. The Web, in contrast, is unstructured and mixes unrelated information, requiring search engines and/or techniques such as the semantic web [23] to locate information in it. For many applications, a more structured approach may be useful—particularly if agents are to make sense of the environment instead of people.

The Web is based on a client-server model. A Web page is stored on a server, and is retrieved (downloaded) by a user program called a client. Content is no longer controlled by the data owner when it leaves the server. As a result, protecting confidential, copyrighted or other sensitive information on the Web is a challenge, except possibly when the client system runs "trusted" hardware and software as has been attempted with some digital rights management systems [1]. At the same time, clients are typically dependent on the interface provided by the Web designer in how they can search remote information and obtain results. In short, for both sides, it is hard to control how data is selected, organised, or presented.

Mansion does not use a client-server based model. Instead, code—in the form of a mobile agent—is shipped to the server to search for information[1]. This can (but need not always) lead to improved efficiency compared to downloading and searching data at the client side. Because an agent runs on a system controlled by the data owner, it effectively runs under control of the data owner. Mansion thus takes an opposite approach to the Web, allowing data to be searched where the data is, and interactions between agents to take place there. There are specific use cases where searching information ''at the source'' is useful:

- It decreases the need for implementing specific predefined server-side interfaces and algorithms to search information and to organise and present results. Although it may be useful to define interfaces for common tasks, giving agents direct access to (raw) data gives users more flexibility in processing information to obtain specific results.

[1] A room can be spread over multiple machines to handle the load of multiple agents visiting a room simultaneously. A room's distribution is controlled the room's owner, and is restricted to a set of machines owned and/or fully trusted by the room's owner (chapter 3). From the point of view of an agent, a room's distribution is transparent and invisible.

- For efficiency reasons, searching raw data may sometimes be better done at the source than, after downloading, on the PC of a regular user.
- Agents can process privacy sensitive information which would otherwise not be accessible. For example, medical data may reside in multiple hospitals that contain textual patient records, lab results, medical images, DNA data, etc. Such data may be identifying and would normally not be accessible. Mobile agents can search such data to, for example, find patients with a rare disease for a clinical trial or for epidemiological research.
- Similar privacy-related issues may stand in the way of deploying centralised (search) solutions for, for example, searching criminal case data stored in police departments.
- In Mansion, data may not have to be exported or made remotely accessible, since organisations can keep ''their'' data in-house and agents may go there. Carefully controlled procedures for exporting information selected by mobile agents may provide a solution to legal, security, privacy or ethical issues that are often associated with disclosing or exporting privacy-sensitive information outside an organisation.
- Books or documents in libraries may be copyrighted and thus not available on the Web. In Mansion, with searching at the source, potentially interesting copyrighted material (e.g., pictures, or music), can be searched before buying it. In addition, very large data collections—for example, containing movies, medical images, etc. may simply be too large to ship over the network. Searching or preprocessing may need to be done at the source for that reason.

The above cases illustrate that it can be useful to search information at the server side instead of downloading it to a client machine. Such cases are the starting point for Mansion's mobile agent based design.

In summary, mobile agents are active, mobile programs that navigate a world autonomously. Agents can search and find information or meet other agents to do business. They can communicate with other agents in a world. This way, agents can interact directly with each other to speed up their search by exchanging relevant information, or to negotiate package deals or contract terms. At the same time, when the need exists, interaction of agents with their environment can be constrained by the system. Example security mechanisms are the application of access control rules to determine whether agents can access particular rooms or content, or (temporarily) disabling communication mechanisms while an agent accesses security or privacy-sensitive content. At the same time, Mansion supports legacy programming languages to ensure that agent owners have the largest degree of flexibility when writing agents to support their task.

Besides exploring applications and the means to structure worlds and control their content, this dissertation also explores low-level (operating system-level) and middleware mechanisms required to support the framework. Security mechanisms are needed to protect hosts

and data, as well as agents and the information stored in them. For example, the mechanism that Mansion provides to control mobile agents when they search sensitive information, entails that agents may not be allowed to communicate with the outside world. This shows that agents should not be able to connect to any party except Mansion's middleware system implementing the *confinement policy*, illustrating why code needs to run under control of the data owner.

1.3. Why agent mobility?

A starting point for Mansion's design is that it should support mobile agents. In fact, a design decision was made that agents *must* migrate physically to a system where a room resides (the room's zone), before they can access that room's content. This section motivates this design constraint.

These are the motivating reasons and scenarios for physical mobility:

- Content in a room may be **valuable** in an economic sense. Protecting intellectual property—such as music files—is a challenge in the Web, since accessing files in the Web means downloading them to a home computer, which makes them vulnerable to theft. In Mansion it is possible to keep files located and accessible on only a few servers. Letting agents access the files to search relevant content, only having selected information exported to a user, may make it easier to limit and control dissemination of content.

- Content in a room may be **sensitive** in a legal sense. Privacy sensitive medical data, such as medical images, may not be distributed freely because of legal or ethical constraints. However, it may be beneficial when medical researchers can access data to find patients with a specific rare disease, in preparation of a retrospective study or for a medical trial, for example. Allowing agents to migrate to data to search it locally to find matching patients, without being allowed to distribute the data, provides a solution to a concrete problem. It may allow researchers to find eligible patients for a trial, involving the doctor to ask these patients to enlist when relevant.

- **Efficiency.** If a room contains a lot of data that needs to be inspected by a program, it may be inefficient to first transfer all the data to a home machine for inspection. Often, it makes sense to (pre-)filter the data at the data's location, and to ship data home only after filtering. Agents can do the filtering themselves, possibly in a customised way. Note that there is a tradeoff here: if computation is expensive compared to shipping the data, especially if many agents do computation over some dataset at the same time, it may be better to ship the data out over the network to the client machine and to let the interested party do the computation there[2].

[2] Note that there may be some Elastic (cloud) solutions which are usable to increase computational capacity under load; these can be deployed as long as these machines can be considered trusted from the room owner's perspective (chapter 3).

- **Flexibility.** Many databases that contain large amounts of data, provide an interface for searching this data, for example as a Web service. These interfaces are either based on prior indexing of (meta)data, or on an algorithm that searches raw data on the fly, as is the case with Web services for searching (matching) annotated genomic data [103]. Both prior indexation and filtering are inflexible. For example, it is unclear whether generic tools can describe the contents of images sufficiently well for all tasks. Mobile agents instead search raw data, implementing a search algorithm most suitable to the task at hand. Although searching raw data may be inefficient compared to approaches that use prior indexation, agents may execute a specific task more precisely. As computing speeds increase, having agents search raw data at the server side becomes an increasingly realistic option, even if these agents consume considerable amounts of CPU time and memory. Especially for research, flexibility and end-user customiseability can give mobile agents an advantage over Web services with hardwired filtering, searching, or indexing routines.

Besides the above motivating examples for agent mobility in Mansion, other motivating reasons for mobile agents were explored in previous work [28,69]. These are slightly out of scope for Mansion, but are discussed briefly below.

Disconnected use. A classic motivation for using mobile agents is that executing an agent's task does not require online availability of the user behind the task. For example, one can submit an agent into the network from a PDA or mobile phone, board a plane from Europe to the U.S., and after the plane arrives seven hours later, collect the agent with its results. A similar idea may be where a user submits a task from a PDA which is too heavyweight to execute on a mobile phone. Although Mansion can support detached usage scenarios, these are not the primary motivating examples for the design, and are not explored in depth in this thesis.

Privacy. Another motivation for using mobile agents is that they can enhance privacy of users. An often-used approach to identify users on the Web is to track them means of their IP address or other identifying *location-bound* features. Mobile agents may hide a user's physical point of presence on the Web. Agents can form a *"mobile proxy"* that allows a user to connect with services through a continuously changing IP adress. When combined with other privacy-enhancing or anonymisation techniques, anonymising ''proxy'' agents may provide a solution to enhance online privacy. However, to achieve real anonymity and unlinkability, various side issues have to be considered. For example, depending on the adversary model, having agents phone home may void unlinkability. It is also unclear how an end-user can pick up an agent with its results without the agent becoming linkable, and log files of various systems may also permit reconstruction of the a user's interactions. Still, protecting end-users against adversaries who are not omnipotent may achievable using mobile agents. Privacy may thus be a motivation factor for using mobile agents. Note that Mansion can provide a way for *pseudonymisation* of agents when they enter Mansion (Secs. 3.10, 9.1.3). The use of privacy-enhancing techniques in Mansion has not been studied in depth.

This dissertation describes the architectural model and the implementation of the Mansion system. Before listing the contributions of this thesis, related work is described.

1.4. History and related work

The first mobile agent system, *Telescript*, was a commercial system with an interpreted language using which agents could be programmed to travel from one physical ‘‘place’’ to another [126]. After Telescript, in the ’90s, a number of mobile agent systems were conceived [19,92,87,55], each specialising in some form of mobile code support, mobility mechanism, security principles, etc. Before describing mobile agent systems, we describe some earlier work in distributed systems that laid a foundation for mobile agent system development.

1.4.1. Mobile code systems

Many technical issues related to code mobility have been explored in the context of distributed operating systems [113,13,29], about a decade before the advent of mobile agents. The main goal of code mobility in the context of distributed operating systems is to migrate processes from one machine to another transparently, usually to facilitate load balancing or fault tolerance.

Examples of distributed operating systems (DOSes) are Chorus and Amoeba [113]. A DOS can be deployed on multiple physical machines, while giving the impression of working on a single machine. To a user, a DOS typically looks like a single Operating System (e.g., UNIX). Processes in a DOS could be placed anywhere on the underlying set of physical machines. When a DOS supports process migration, processes are restarted at the same point in execution as before migration. This is often referred to as **strong migration**. Strong migration can be awkward to implement; a process is migrated and restarted transparently, together with its internal state including registers and stack pointers, at exactly the same point in its thread of execution as where it left off before migration. A problem is that after migration, references to local resources (e.g., open files) may have to be remapped, to point to a local resource or to refer back transparently to the original resource, depending on the resource and DOS implementation. Dealing with signals and timers is also difficult.

Weak migration is the opposite of strong migration. A process starts at its program’s initial entry point (main) after migration. Using weak migration, a process (i.e., its programmer) is responsible for packaging the process’ relevant state before migration, and for unpacking it after migration; nothing happens automatically [69].

Strong migration is convenient from a programmer’s perspective. For example, a process may keep an implicit list of ‘‘visited sites’’ on its stack as the result of a recursive algorithm that visits multiple rooms, unwinding the stack and travelling back automatically after hitting a dead end. Such an algorithm is harder to implement using a weak migration model, as the

language's (implicit) support for recursion by pushing data on the stack cannot be supported or easily as the stack and other runtime information is destructed at migration time. Weak migration is more convenient from the perspective of the system designer, as in this case he does not need to worry about the issues that were described for distributed operating systems at migration time. Note that most distributed operating systems were running on homogeneous hardware; for distributed (operating) systems running on different hardware architectures, implementing strong migration may be infeasible, particularly when supporting compiled binary programs.

Most traditional programming languages, for example compiled C, C++, Java (or the JVM as a whole), or Python, were not designed to support (runtime) process migration. This is particularly true if a program is to be migrated over different CPU architectures. Java or other interpreted languages built upon a virtual machine can theoretically support strong migration more easily than compiled binary programs can. However, for Java-based agents, mobility support requires modifications to the underlying virtual machine [14, 25].

Distributed operating systems that support strong migration, have to manage issues like capturing and reconstructing operating system-specific process state—such as open file descriptors and alarms—after process migration themselves. These problems can often be solved relatively straightforwardly, since distributed operating systems provide a controlled environment that is designed specifically for tasks like load balancing and process migration. A particular advantage of distributed operating systems is that these systems usually are distributed over a limited set of homogeneous resources on a local area network, each running an identical distributed operating system image. From a security perspective, the advantage is that a DOS is deployed within a single trusted administrative domain. each running an identical distributed operating system image.

Unfortunately, the realities of Internet-based distributed mobile agent systems are much more harsh than the homogeneous, single-administrator world that underlies these distributed operating system environments. Providing a truly transparent strong migration model is much more challenging to maintain (if not practically infeasible) in more heterogeneous Internet-based systems than in distributed operating systems [69, 112].

1.4.2. Internet-based, heterogeneous distributed systems

Internet-based systems are systems that are spread over multiple administrative domains spread across the Internet. Internet-based systems have a far-from-uniform underlying substrate, in terms of hardware and software stacks. Machines in different administrative domains differ in terms of hardware (e.g., CPU type, speed, and load) and in terms of operating system, configuration and administration; that is, this environment is extremely heterogeneous. Network speed, reliability and reachability (the ability of a given host to reach a given other host anywhere on the Internet) are neither guaranteed nor uniform on the Internet [26]. As a result, many of the assumptions applicable to distributed operating systems cannot be

applied to Internet-based systems. Also, the security requirements of current-day Internet-based agent systems are often more stringent and also more complex than for the single-administrator, trusted-hardware model that underlies most distributed operating systems developed in the '80s. The following sections describe some solutions that have been conceived in the context of heterogeneous Internet-based distributed systems.

1.4.3. Computational grids

A conceptually similar, but often different and often much more simplistic, descendant of distributed operating systems are computational grid systems, such as Globus or gLite. Globus is a minimalistic middleware, or toolkit, that runs on a large set of cluster computers all over the world. Grid middleware makes resources all over the world available for (scientific) applications. In a sense, the grid can be seen as a giant cluster computer, except that it is heterogeneous and that the network speeds, especially between clusters, can vary widely. Globus allows distributed (supercomputing) tasks to run on computing resources owned by various organisations all over the world. These computing resources are typically cluster computers running Linux, augmented with various components of the Globus toolkit which allow for, among other things, submission of jobs (programs) into the cluster. Typically, storage and computational resources are owned by virtual organisations (VOs), and users must be authenticatable as members of such a VO before they can submit computational *jobs* to the resources owned by a VO [9, 38].

An extension of the Globus middleware that attempts to provide a more concise view of the ''cluster world'' which underlies a grid, is the *gLite* [59] middleware—a middleware system based on the Globus toolkit, built by a team of researchers from CERN and various contributors. The difference between gLite and Globus is that gLite comes with a set of overarching services, such as job submission services using which researchers can find their way to any cluster in the gLite grid. Other examples of global services provided by gLite are file and resource naming (lookup) services that allow users or jobs to locate resources in the system.

Globus or gLite do not provide a single ''Operating System'' view to applications. The grid does not hide distribution aspects from applications. Grid systems are implemented by a middleware system running on top of standard operating systems, that do not support process migration, certainly not strong migration. Programming or using applications on the grid is often rather low level and in many cases requires specific handling of errors due to failing jobs or incompatibilities on different machines, although some workarounds exist [44, 75].

Globus (and gLite) are somewhat minimalistic by necessity; these systems have to run on a variety of cluster machines, each with different configurations and different administration, on which they have to permit compute jobs to run in as efficient and as simple a way as possible. The intended user community of a grid consists of scientists, who are supposed to be able to handle considerable complexity and to invest the effort needed to get their applications running in the system.

Programs submitted (as jobs) into the grid are not validated or signed for security; the grid VO administrators trust their users not to submit malicious code into the system. The relative trustworthiness of the—typically small—set of scientific users that use a VO, keeps the security model of grids tractable. However, this trust model is also a weakness of grid systems. If a VO's systems are used by many users and also used for running privacy-sensitive tasks, one of the users may take advantage of a vulnerability of the system and gain access to sensitive data of another user using a malicious job.

In contrast to grids, Internet-based mobile agent systems may be deployed over a large set of arbitrary and a priori unknown machines, owned by arbitrary users. Agents may be submitted by any user, and may consist of arbitrary (unsigned) code. The intents of users may range from running a simple computational or collaborative task, to launching a distributed denial of service attack, to spamming all citizens of all industrialised (or nonindustrial) countries in the world. The difference between grids and mobile agent middleware, thus, lies in the scale and the number of users and the assumed trustworthiness of users in the relatively controlled environment of computational grids.

Whereas most grid systems use ordinary security controls based on the traditional UNIX user-based discretionary access control models, with a relatively high degree of trust in the users and in the programs that these users submit, such a ''trusting'' security model does not suffice for very large-scale mobile agent systems where we can make little or no assumptions on the users who can inject agents into the system, or their intent. In short, the grid *trust model* does not scale to general Internet based systems.

1.4.4. Desktop grids

Desktop grids (DGs), such as *XtremWeb* or *BOINC* from the University of California at Berkeley [34,10] take a radically different approach to providing infrastructure for large-scale scientific computations than traditional grids. Desktop grids are designed to ''harvest'' CPU cycles and resources of (idle) machines within an organisation or on the Internet. On participating machines, a middleware system that fetches jobs is run by the owner. As long as the owner of the machine does not need its resources, the desktop grid job may run on it—often, the DG environment runs as a *nice* process (i.e., at low scheduling priority, invoked using the `nice` system call), or as a screen saver. Another name for using idle time of contributing computers on the Internet is volunteer computing.

Desktop grids do not accept arbitrary programs from arbitrary users on the machines they manage. Instead, in most cases, the potential programs are screened and/or programmed by trusted administrators of the desktop grid, possibly signed by them, and only after the jobs are sufficiently vetted may they be submitted to the desktop grid. Desktop grids can be seen as a somewhat more flexible version of *SETI@Home.*[3]

[3] http://setiathome.berkeley.edu/

Condor [114] is a more general-purpose version of a desktop grid, intended to be managed in (large) organisations where sets of machines may be idle for large periods of time, for example at night. Some hospitals and some universities run ''Condor pools,'' sets of machines that run the Condor middleware. In Condor, typically, only trusted administrators submit (signed) jobs to run in the Condor pool. The Condor middleware accepts only authorised jobs, to avoid that (potentially sensitive) files or resources from the hosting organisation become corrupted or stolen, although this risk still exists—how does a Condor administrator vet the code that a researcher wants to submit into the pool?

An advantage of a Condor pool compared to a desktop grid, is that it runs in a relatively small environment, typically a single organisation. This may allow trusted local users (e.g., researchers) to use the Condor pool. Thus, depending on regulations and deployment, it may be possible to submit computations and sensitive data into the pool, as the pool runs in the researcher's organisation. Note that many hospitals to date also maintain a small cluster computer, which may be considered safer for some (sensitive) scientific applications.

An interesting distinction between (desktop) grids and mobile agent systems, is that in a desktop grid, users typically submit a (trivially parallel) ''bag of tasks'' that should be run on a set of remote machines in the grid, typically each with their own piece of input data. This is a form of ''remote evaluation.'' After a task is completed, the user (automatically) obtains the results. Mobile agents, also called *itinerant* agents, take a different approach. In contrast to desktop grid applications, which are often regular applications executed by a script, agents typically (autonomously) navigate through the system. This navigation ability has implications for the way in which applications are programmed (agents need to be location and mobility-aware), and for the abstractions provided by the middleware [52]. Security issues (e.g., related to safeguarding data stored in mobile agents, authentication, and protecting hosts on a multihop itinerary) are discussed throughout this thesis.

1.4.5. Internet-based mobile agent systems

As mentioned before, *Telescript* was the first mobile agent system. It featured an interpreted language using which agents could be programmed to travel from one physical ''place'' to another [126]. Telescript featured strong migration, using an interpreted, object-oriented language similar to C++. A specific operation in Telescript was the *meet* primitive, which allowed agents to colocate at some known place and time. Colocated agents could invoke methods on each other—for example, where one of the agent implemented a service. This provides an alternative to (generally slower) remote invocation, as colocated agents can interact in real time without network delay. Telescript was a commercial system and is no longer in production.

A canonical example of a mobile agent application is travel planning. Travel planning must be tailored to the needs of a specific user, who wants to plan a specific trip. In a travel planning application, a site accepts programs that are written by users for the task of planning

an itinerary, and possibly for booking the trip. Executing the task requires migration autonomy, as the agent must be able to visit sites of different travel agencies, airlines, hotel booking sites, etc., to find the cheapest offer meeting the agent owner's requirements. Also, security is imperative: the agent should be protected against tampering and cheating, for example to avoid that it thinks that it is getting the cheapest deal while in fact it has taken the most expensive offer.

Internet-based mobile agent systems have a different underlying systems model compared to the distributed systems described before. If mobile agent systems aim at Internet-scale deployment, issues such as portability and security arise. Mobile agents can generally not assume to be running on a (relatively) trusted set of cluster computers or on some homogeneous set of machines in a local network, as agents go out to visit remote machines containing interesting content that may be located anywhere on earth. As with desktop grids, this may include machines owned by regular users or companies on the Internet, which cannot be trusted in advance. In contrast to desktop grids, mobile agent systems should allow for execution of *arbitrary* programs, generally written by arbitrary (untrusted) users.

Agents may be customised or implemented by users to embed the search strategy or intelligence they consider useful. Mobile agents can be tailored to the needs of a specific user. Flexibility (of implementation) is a primary design goal. Due to the flexibility-of-code requirement, it is not feasible to verify or vet all possible mobile agents, or to make any a priori assumptions on their correctness or intent of the agent—particularly in a very large-scale environment—as may be feasible in desktop grids. There may not even be a central entrance point through which agents are submitted. In short, few assumptions can be made regarding the trustworthiness of people, content, agents, and hosts that are involved in designing, writing, and deploying mobile agents, mobile agent systems, and content.

The primary focus of most mobile agent systems—the majority of which designed in the second half of the 1990s—is to achieve end-user agent programmability and flexibility while meeting the goal of portability (being able to run agents in a heterogeneous environment), and security. The approach taken in most cases is to support only one (relatively portable) programming language, usually Java. Java is often chosen because it provides code portability as well as security by *sandboxing* interpreted code running inside a Java virtual machine (JVM), although standard Java does not provide a way to serialise the state of running threads (for strong migration). Other examples of interpreted (portable) languages used in mobile agent systems are Safe-Tcl, Scheme, and Python [127, 29, 63]. Some systems support (interpreted) C agents. Typically, some form of code signing is used for security [91]. Code signing is rather inflexible and primarily useful for binding agents to identities.

Despite years of (research) work, no real ''industry standard'' agent system has emerged, particularly not for mobile agent systems. JADE, a Java-based *multi*agent system backed by a consortium of telecom providers, is one of the few agent systems in active development to date that has a significant deployment base. JADE is however mostly focused on stationary agents, with different agents executing (part of) a task, and it has only limited support for agent mobility. The system's main function is to allow agents that run on different

machines within a closed system to collaborate [21]. Security extensions called JADE-S exist that provide agents with mechanisms for authentication and authorisation of other agents, containers (execution environments that agents run and invoke services in) and services, based on decentrally managed delegation certificates [94, 61]. However, these mechanisms alone do not suffice to consider JADE secure when considering untrusted or malicious mobile agents or agencies in a distributed context [32, 119]. Some work has been done to support (selective) encryption of content of mobile agents in JADE, [46], but these have limited usability as the overall security model for mobile agents is incomplete. For example, integrity protection of agents (i.e., audit trails) or means for authenticating and expressing trust in *agencies* (the middleware on a host that receives agents) are lacking.

Mobile-C is a mobile system that is, like JADE, compliant with the Foundation of Intelligent Physical Agents (FIPA) standard [21] agency-model [28]. Mobile-C is specially designed to be used in (closed) factory automation systems, where all agencies and the agents in them are essentially trusted. The main threats considered here are external threats, e.g., where an outsider tampers with an agent while in transit. The framework makes use of ssh so secure agent transport, where known_host files are distributed between all peers to establish mutual trust. The system is not designed for large-scale use. The C agents are not compiled binary programs but run in an interpreter.

Notable security work is done in the SeMoA mobile agent system [99]. SeMoA uses a specially constructed Java agent server that applies various security techniques. For example, it can apply filters to incoming and outgoing agents, which allow for authentication and conditional encryption or decryption of content and for constructing basic audit trails. SeMoA applies various techniques to prevent agents in a virtual machine from invoking objects instantiated by other agents. Because SeMoA is constructed using a regular JVM with unmodified packages, however, it suffers from inherent shortcomings in the Java security model such as insufficient protection against a number of DoS attacks that can be mounted from within an agent, including memory exhaustion [99].

Another agent system that is currently in active development is AgentScape [127]. AgentScape is a mobile agent platform initially developed at the Vrije Universiteit Amsterdam[4]. Work done in AgentScape includes resource management, service composition, and agent matchmaking and clustering [72,84,83]. Agent servers (currently supporting Java and Python agents) manage agents and provide them with an API. AgentScape has many similarities to Mansion, and parts were co-developed. AgentScape uses the Agent Operating System described in this thesis (Ch. 4). It makes use of its agent container integrity protection mechanisms and it can also use audit trail based security mechanisms similar to those presented in this thesis [116]. In contrast to JADE and Mobile-C, AgentScape is designed for large-scale distributed use; it shares many of Mansion's design goals. Apart from implementation, Mansion is distinct from AgentScape in the paradigm it provides and its use of jailing (Ch. 6) to manage host protection in a language-independent way.

[4] AgentScape is currently supported by and maintained by a consortium, led by TUDelft and Thales NL.

The prevalent way to approach security in mobile agent systems is to force agent programmers to use one or a small set of programming language(s) that match available agent servers or containers. This approach has disadvantages, as it a priori excludes the use of other languages that may potentially be better-suited to perform a given task. It also precludes the use of legacy programs—often ready at hand to implement some task—in or by mobile agents. It is possible that the restriction in choice of programming languages, particularly the lack of support for ''legacy'' programs including scripts and compiled C programs, is a cause for mobile agent systems not having taken off at a large scale in practice, as opposed to, for example, grid systems that readily support (existing) programs written in any language that is supported by the underlying (UNIX/Linux) system. There exists an enormous amount of efficient and ready to use legacy code, implemented in various languages (think also of, e.g., perl), sometimes available only in binary form. Software engineers may avoid solutions where the reuse of existing code is precluded.

1.4.6. Advancing the state of the art

In 2004, an overview of some possible causes for the lack of uptake of the mobile agents paradigm was given by one of the authors of TACOMA [50]. The proposed causes included security being left as an unsolved problem in most systems, a repeated focus on Java-based agents in most solutions, and also the lack of a ''killer application'' and the inability of the mobile agent community to ''converge on a standardised protocol and programming model for Internet applications using [agent] technology.'' Whether or not these claims are right, Mansion addresses and overcomes some of these issues.

Mansion presents a new programming paradigm for mobile agents, centered around closed, scalable worlds that contain hyperlinked rooms. Mansion provides a *structured* environment for mobile agents, where the structure is application-dependent and defined by a world's designer. The paradigm will be explained in Chapter 2. Support for applications—centering around access to sensitive information—that are hard or impossible to solve without mobile agents will be presented in the next section.

Most security problems common to mobile agent systems are addressed. This thesis presents solutions for host protection, protecting content (using access control), and protecting an agent's state against tampering. It also presents control mechanisms that include controlling membership of a world and means for global resource usage accounting and limitation. Interoperable protocols and APIs for (portable) migration and audit trail construction have been implemented and tested for interoperability and performance. A low-level jailing system provides a portable way (among UNIX systems) to securely execute mobile agents written in different programming languages, including binary agents, in a confined environment that can be controlled using a simple, intuitive policy model. Combined, many of the problems of mobile agents that were discussed in in [50] are addressed by Mansion.

One particular security problem is not addressed. Protecting an agent against a malicious host that tampers with the agent's code at runtime is infeasible in practice and outside the scope of this thesis; we assume that agents trust hosts that they migrate to, and any potential malicious alterations to an agent cannot spread from the host that these are made on. Although approaches are known to exist to help protect agents against this threat for at least a limited period of time (e.g., code obfuscation, see [48]), these are not generally applicable to all existing programs or programming languages.

Mansion can detect tampering with an agent's state after the fact, using audit trails (Sec. 8.3.1); this provides a powerful deterrent against misuse. Based on these, an agent's owner can determine whether a given agent has been tampered with on a particular machine. The presented approaches support reputation-based mechanisms for trust management and, practically, to prevent agents from migrating to hosts that may have tampered with agents. Because agent code is *signed* for integrity, and because an agent is restarted each time when migrated, tampering with an agent is limited to execution/runtime modifications on their current machine, and cannot spread to subsequent machines.

Protecting agent's state is important because data may be falsified in mobile agents. (Commercial) sites that receive agents may have an incentive to cheat, for example, by giving an agent a low quote to make it decide to buy something, while the real price is significantly higher. There may be an incentive for hosts on an agent's itinerary to tamper with an agent's collected state from an earlier host (a *hop*): a host may up the quotes obtained on earlier hops, to make its own (high) quotes look better. We prevent the latter attack by ensuring a secure, tamper-evident audit trail of all changes made to an agent throughout its itinerary. Combined with knowledge of the agent's decision making algorithm and the amount of money actually spent (the decision made), an assessment can be made whether the agent was cheated on a particular hop. Such verification can take place probabilistically: an agent's owner may occasionally go through a returned agent's audit trail to recalculate all intermediate results, based on the offers that were collected by the agent to establish that the agent indeed got the lowest offer on a given machine and was not tricked into it.

Detection of fraud can have direct repercussions for the fraudulent host (i.e, removal from the world) or, if evidence is not hard enough, may result in its reputation decreasing. Reputation-based techniques may be implemented by agents (or their owners) exchanging information about possibly fraudulent hosts, which can result in agents avoiding machines with a large ''fraud probability.'' Reputation-based techniques may be applied at the application level and are outside the scope of Mansion. Protection of resources is further achieved by access control and other approaches that will be presented in later chapters.

1.4.7. Confined agents

A particular contribution—an attempt to address the lack of a ''killer app'' for mobile agents—is that of agent confinement. The paradigm provides a special logical construct

called a confined room. A confined room is a special room that agents can migrate to. In this room, agents can search for information but they cannot take anything out: no content, no notes, no memory. Agents are restarted with each migration (Mansion implements weak mobility), so they retain no memory when they leave a confined room. Further, agents in a confined room cannot communicate with agents outside the room, and they cannot store anything to keep with them as they leave, as would be allowed in a regular room.

The idea of confined agents is that they can safely search confidential or other sensitive information in a confined room—they execute on a system of the data owner, so the data owner can be sure that the room's restrictions are applied—without risk of this information leaking out. If the agent has found any information that it thinks might be interesting for its owner, the agent must pass this information to a special ''guardian agent'' provided by the room's owner, that inspects the information or request, and if allowed, provides the agent with a means to obtain selected information after it left the The way in which information is exported and under what conditions (payment, nondisclosure contract, etc.) is determined by the room owner.

The main added value of entering a confined room from the agent's perspective, is that the construct allows them to access information that may otherwise be inaccessible (due to e.g., intellectual property, financial or privacy constraints), and search it before selecting data for buying or otherwise obtaining it afterwards.

Conceptually, being in a confined room is a bit like being in a library containing sensitive documents. One can enter—if authorised—to read the documents, but one is not allowed to take any notes or copies of what is seen. If one wants to take information out, one needs ones notes to be vetted by the library clerk. A special property in this analogy is that one will be brain-wiped by the clerk when leaving because the library owner is paranoid that one has photographic memory and may pass on even the slightest bit of information. Only information explicitly vetted by the clerk may be exported, and often only in original format (your notes will be destroyed as the clerk is worried about steganography). In short, selected data is exported only in a way deemed safe by the library owner, possibly after signing a contract or paying for the information; nothing of the agent's original search remains after it left the confined room.

Although some earlier work described confinement, that is, searching under control of the data owner, as a possible selling point for mobile agents [100,22], these approaches focused mainly on intellectual property protection, where some compressed information on results (e.g., thumbnails of images) could be returned to clients to be used for a decision to buy the information. This thesis is the first to present confinement as a solution to search privacy sensitive medical data which provides a stronger form of confinement, where manual checking or, possibly, contract negotiations may be required before *any* result is returned, although weaker confinement regimes can also be applied. Further, it is the first to present confinement as a generic *logical* programming construct that is visible to agents as part of a programming paradigm.

1.5. Contributions of this thesis

The contributions of this thesis are broad.

First, a paradigm for modelling and structuring applications is presented, using the concepts of application-specific worlds that contain hyperlinked rooms where agents can go to meet or find content (chapters 1-3). This paradigm allows for development of distributed worlds for different applications. The thesis explores some applications that can be supported by the paradigm and describes a prototype application (chapter 10).

Second, an application programming interface (API) and a middleware design are presented that support the paradigm (chapters 4-9). The middleware addresses practical problems, such as the problem establishing audit trails that cannot be tampered with, and support for (compiled) legacy code that has not been solved by other agent systems. The middleware system has a modular design. Among other things, the thesis presents a common communication substrate that allows for secure (distributed) deployment of the system. This substrate, like other components of the Middleware, is reusable in other distributed (mobile agent) systems. The various components, protocols, and concepts that compose the Mansion system presents (sometimes incremental) novelty.

Third, the confinement model, presented in the context of Mansion, provides a novel approach for maintaining control over export of information, while permitting freedom to agent owners to implement programs and algorithms that best suit the task at hand (chapter 2). A concrete, real-world scenario involving privacy-sensitive medical images that are made available in different rooms by different hospitals, to allow for search by (confined) agents of researchers while allowing for tight (manual) control over the export of information, provides a concrete use case for this model (chapter 10).

Fourth, the design of the paradigm and the middleware system that supports it is a contribution in itself (overall thesis, chapter 11). The design process involved a careful tradeoff for every architectural decision, to ensure that the distributed system and the components in it provide sufficient control mechanisms so that the system can be effectively protected against misuse, faults, malicious host, rogue agents, etc. in a scalable way (also from an administrative perspective), while at the same time ensuring that content owners are free to autonomously add systems (computers), information, rooms, hyperlinks and other content, and that agent programmers are free to use whatever programming language or algorithm they like and can go to any room they like without being tracked or otherwise controlled[5] by the world owner or some central authority—in short, while designing the system we took care to balance and maintain controllability, scalability, security and autonomy of the system. By design, the model allows different independent trust models (from the perspective of world owners, content providers and agents, respectively) to be mapped on a given system.

[5] An agent location service which tracks agents is designed, but there may be multiple of these services, each managed by a different authority, and selected by an agent owner on world entrance.

1.5.1. Research question

The Mansion paradigm and its validation presents the main contribution of this thesis. Overall, the following research and design question drove the Mansion design:

Can we design a secure distributed system that can be structured such that content can be effectively and flexibly located by mobile agents, which balances security, scalability and controllability concerns of world and content owners on the one hand against the need for autonomy of content owners and agents owners on the other hand?

A recurring design theme was the tradeoff between controllability and autonomy. The tradeoff between these requirements occured with almost every major middleware component's design. For example:

- For a world: how to control world membership in a secure way, without at the same time requiring the world owner to control every detail of a world? More specifically, ensuring the system remains scalable (also in an adminstrative sense), at the same time ensuring the world remains controllable *and* preventing the world owner to interfere with a content owner's autonomy to place hyperlinks or content in the world?
- For content and host owners: how to control agent processes such that system security can be protected and confinement policies can be imposed, while avoiding that agent programmers must use a specific agent programming language and avoiding to have to snoop on every instruction an agent program makes?

The question of how to ensure autonomy at one level while providing security or controllability at another level is a generic one that touched every aspect of the system's design—from low level aspects to the design of the paradigm.

1.5.2. Dominant design requirements

From the above, we selected the following high-level design requirements to drive the design:

- Conceptual clarity. The system needs to provide a clear programming model using which agents can find information effectively, within a potentially large-scale distributed system. It should be possible to customize worlds to a specific application's needs and express the world's structure, content and rules to agent programmers, such that programming agents to find information and/or interact with other agents is doable.
- Security. The system needs to address security issues, like how to protect hosts against mobile agents and vice versa, without imposing too many constraints on the agent programming model, preferably with supporting legacy and binary agents.

- Scalability. It should be possible to manage membership of a world effectively, including the ability to add and remove systems, and ultimately content, from a world, when such systems or content do not adhere to the rules. while ensuring that the underlying system remains manageable and scalable. When load increases on (parts of) a world, the system should be extensible with hosts and (replica) content to cope with this load gracefully. Administrative scalability needs to be taken into account: an increase of load in a given room should not involve action of other—or worse, central—administrators; similar for removing erroneous hosts or damaged content.
- Autonomy versus controllability. The design should allow people to create, manage and add or remove content and agents autonomously, without central control over all content or every detail. Security and controllability should be possible at the world and other levels, while content and agent owners' autonomy (and privacy) should not, or in the least possible way, be violated. Given this tradeoff, world designers should be able to influence the decree of control needed and possible.

The Mansion design is the result of a design process that took the above requirements as leading. The result: the Mansion paradigm, the programming interface and the middleware system that supports it, is presented in this thesis and summarized below.

Mansion provides a paradigm to structure mobile agent applications. The paradigm provides closed, application-specific worlds to allow world designers to create worlds specifically for their (distributed) application. Agents can be programmed specifically for a given world. The logical structure of a world should allow agents to be tailored precisely to the application for which it is built; the Mansion logical model for structuring an agent's environment allows doing so. The prototype world demonstrates that, given a simple hyperlink layout, it is straightforward to program an agent to search the world.

From a distributed systems perspective, the system should be secure, efficient, sufficiently reliable (i.e., making sure that an error in one component can bring down a whole system), and it should be scalable. Scalability concerns how a system can cope with increasing numbers of machines, administrators, and content, possibly spread all over the world, and also with large numbers of agents visiting a world. A system needs to provide fairness from the perspective of components and resource usage, as well as from an administrative perspective: the burden of administering new rooms, content, users, agents, etc., should be spread more or less equally over the participants of the system and not be placed on, say, a single administrator or component of the system. Scalability issues were not tested extensively in our research—we tested a prototype world which ran on the DAS3 cluster computer, but have not deployed the system at large scale—but are addressed throughout the design. Scalability aspects are described specifically in Chap. 3.

Sometimes, requirements are in conflict. For example, consider replicating data for availability, versus the need to limit data distribution for security/privacy reasons. Reconciling conflicts can be difficult. The Mansion conceptual model often provides clarity that helps

resolve such conflicts. For example, administrators can replicate data, but not outside a *zone*, which is a set of machines typically owned by a single organisation. The paradigm is designed such that programmers and users of the system can have a clear understanding of the system's properties, while hiding normally unnecessary details such as related to physicial location except when asked for it (e.g., an agent normally only sees a logical hyperlink to a room, but can enquire about administrative or security properties of the system that it points to if required).

1.5.3. Technical contributions

In addition to presenting the overall design as outlined above, the thesis presents the following concrete technical contributions:

- An overarching security architecture that protects the system, including agents, content, machines and users, against malicious parties or components, at least to the extent that misuse can be detected and malicious agents or components can be held or removed from the system.
- A novel jailing system for confining (compiled, binary) agents. The jailer runs in user mode in Linux, and can in principle be ported to other (UNIX) operating systems. Jailed processes can only access resources (like files or Internet addresses) if a policy explicitly allows this. Results show that arbitrary programs (including scripts and legacy programs) can run efficiently in a jail despite being sandboxed.
- A portable component has been designed and developed for constructing middleware systems, called agent operating system (AOS). AOS provides the functionality with which storing and shipping agents between middleware processes in a secure and language-neutral way. AOS has been demonstrated to be portable, interoperable, and usable in different middleware systems [77, 78].
- A hierarchical per-world location service infrastructure is devised, whose deployment can be adapted when usage of a world grows. It is implemented using an object type that runs in an object server constructed for Mansion which allows for jailing objects to protect the system against possible malicious objects (e.g., in case future agents can place objects in a world). Mansion is designed such that this can be done without requiring centralized control over all components or content of a world.
- A location-independent identifier scheme for objects and hyperlinks that supports authentication of different middleware processes using a single self-certifying identifier. This model allows for transparent replication of rooms and objects while replicas can be authenticated using the same identifier.

- A handoff protocol that ensures that when a host detects tampering with an agent during migration, it is not accepted and the agent's contact address is not updated in the agent's authoritative agent location service.
- A confinement model for mobile agents is presented, using which mobile agents can search data in a room while the room owner remains in control over what information leaves the room. The confinement model is an attempt to resolve the "lack of a killer app" issue that prevents adoption of the mobile agent paradigm according to [50].

Some of the above contributions are concrete stand-alone components.

1.6. Outline of the dissertation

The thesis is structured such that the overall system's description and distributed system (security) architecture is described in the early chapters (describing the model), and the validation is in the final chapters that describe and discuss experiences with the prototype world. The middle part of the thesis describes concrete middleware components, including the jailing system and AOS. The concrete technical components contributed by this thesis are described in chapters 4 to 9. This dissertation is structured as follows:

Chapter 2 introduces the main architectural elements and the logical (conceptual) model of the Mansion programming paradigm.

Chapter 3 describes how the model maps on the underlying physical system (hosts), including logical concepts such as worlds and rooms. Important security concepts used by Mansion are introduced, as well as concepts that are crucial to administrating a closed-off Mansion environment for a specific application.

Chapters 4–9 will describe the realisation of the Mansion middleware. Chapters 4–8 describe the way in which the Mansion middleware is structured and some of its components: the jailer, agent operating system (AOS) kernel, and the Mansion object server. Mansion has a modular, layered design. The above-mentioned chapters describe the design of each component and layer, including performance measurements where applicable. Chapter 9 describes the interface with which agents can do work in a world, the Mansion API, in full.

Chapter 10 explores several scenarios for using Mansion, and describes a prototype world and prototype agents that were implemented to validate the approach. It also reports on some end-to-end performance measurements made when using the overall system. The thesis concludes with a discussion. A summary is given in chapter 11.

1.7. Typographical notes

The following typographical conventions are used in this thesis:

- **Bold** fonts are used for definitions
- *Italic* fonts are used for method and interface names and definitions, and for system calls, and arguments of methods and system calls. Path names and program names are given in regular fonts. Program excerpts are shown in courier typeface.
- Capitalisation is used when referring to a specific system or its name, such as Mansion, the Web, or AgentScape, but not when naming a class or type of system, such as grid systems. Acronyms are capitalised.

.

Chapter 2
The Mansion Paradigm

This chapter introduces the Mansion paradigm. The Mansion paradigm is a logical model that provides the foundation for programming mobile agents. The paradigm is supported by a middleware system, which provides a programming interface to agents. The middleware ties the Mansion logical model to the physical model: the middleware components of which a (distributed) Mansion system consists at runtime. This chapter presents the logical model. The physical model will be described in a later chapter.

First, the architectural elements of the Mansion model are described, followed by how these elements form a Mansion world.

2.1. Architectural elements

The following logical elements are visible to agents:

- **World:** a collection of rooms, connected by hyperlinks.
- **Room:** a place where agents and objects and hyperlinks can be. An agent can be in only one room at a time.
- **Hyperlink:** a unidirectional link from one room to another.
- **Object:** a passive entity, which can be accessed and manipulated by its owner and by agents. Objects may contain data such as files.
- **Agent:** an active, autonomous entity that can inspect objects in a room and which can interact (communicate) with other agents in the world.

All entities in a room are visible using sets of attribute-value pairs called **Attribute Sets.**

2.2. The conceptual model

This section describes the main architectural elements of the conceptual model.

Agents are *autonomous* entities. An agent can decide autonomously with what object to interact, what hyperlink to follow, or with what agent to communicate, within the constraints of the Mansion paradigm. An agent is a (single or multithreaded) process running on one host. Agents can communicate with another agents using secure, bidirectional, reliable, and ordered connections. When an agent enters a world, it receives a unique identifier, *AgentID*. An agent can set up a connection to another agent using that agent's AgentID.

Because agents are autonomous, they may refuse or ignore connections from other agents, and may or may not reply to requests from other agents. Except for communication, there is no way to directly interact with another agent: no entity is allowed to interfere directly with the execution thread of another agent. Agents do not expose an interface for other agents or room owners to invoke methods. Allowing agents to invoke methods on other agents would violate the agent's autonomy and could prove a security risk.

Each room may contain one or more objects. An **object** is a passive entity that encapsulates state, and which can be invoked using an interface. An **object interface** contains a set of methods; the object types, that is, the methods that objects provide, may differ per world. Objects can be invoked only by agents in that room. An object may only be in one room at a time. An object cannot invoke methods on anything. This constraint ensures conceptual integrity: an object visible in one room can only be changed due to actions of entities in the same room; no changes may be made to an object's state as the result of a method invocation by an agent in another room.

Mansion makes a clear distinction between agents and objects. Agents are active entities, which have their own threads of control and can invoke methods on objects and communicate with other agents. Agents are autonomous in deciding if and when to accept communication requests, and whether or not to read incoming data or react to it. Objects, on the other hand, are passive entities. Objects have private state encapsulated by an interface which can be invoked using (synchronous) method invocations. They do not have their own threads of control. Invocations may return data and/or change internal state of the object. A Mansion object is conceptually similar to, say, a C++ object.

Each Mansion application is structured as a world. Different worlds for multiple applications are not interconnected, and can coexist independently. This is fundamentally different from the Internet where, in principle, anything can be linked to anything.

Each world consists of a set of rooms, interconnected by hyperlinks. Hyperlinks are unidirectional links from one room to another room. Agents use a hyperlink to migrate from one room to another. Rooms must be reachable through hyperlinks: a room which is not reachable through some sequence of rooms and hyperlinks is not part of a world.

An entry point of a world is called a **world entrance room (WER)**. There may be multiple world entrance rooms, which are rooms that are specially marked as such by the world administrator and known globally in a world. Different (overlapping) hyperlink structures

may exist in a world; a room may have multiple hyperlinks pointing to and from it. (If there are multiple WERs, there may be multiple roots to a hyperlink structure.) There may or may not be an a priori structure imposed on the hyperlink layout of a world. Hyperlinks, rooms and content may be added at runtime, depending on the world's constraints.

Worlds are closed: a room in one world may not be part of another world and cannot be linked to from another world. Agents enter a world using a process called ''agent injection.'' The result of injection is that an agent is started up in one of the world's world entrance rooms (WERs), typically one auto-selected by one of the **world entrance daemons** (entry points for agents) through which agents may be injected. A possible criterium for selecting a suitable WER may, for example, be a payment scheme, or a specific topic that the agent is interested in[6]. From a world entrance room, an agent may migrate to another room by following a hyperlink. When the agents exists a world, it can be collected from a daemon called the **morgue** by the agent's owner.

A world designer can influence certain aspects of a world, such as the types of objects that exist in a world or a world's hyperlink topology. For example, there may be a way to a given room, but no way back. The structure of a world depends on the application. An agent may only be able to migrate forward through a pre-established path of hyperlinks, with only an occasional detour. In another example, all rooms may be interconnected to all other rooms. Such aspects are described in a **world design document,** which describes a world and is used to configure it. This document is described later in this thesis.

All entities in Mansion are described by means of **attribute sets (ASs)**; using the attributes in a hyperlink's AS, an agent can determine which hyperlink to follow to go to a given room. Attributes also help agents to find interesting objects or agents in a room. An attribute may, for example, contain an agent's AgentID besides a description of its interests. Mandatory and optional attributes in an AS are defined in a world design document; agents are responsible for filling their attribute set when they enter a room; room owners define the AS'es for objects and hyperlinks in a room.

Attribute sets are registered in a special per-room object called the **Room Monitor Object (RMO)**. Each room has one RMO, which contains attribute sets and other information that is needed to make a room work such as a list of the agents in the room and other bookkeeping information; the RMO acts as the registry of a room. An RMO—and all other objects in a room—is available only from a limited set of machines, that each run a middleware process. This set of machines is called a **zone**.

A **zone** is an administrative and security concept, that can be intuitively regarded a set of machines under single administrative control that host a set of middleware processes that can contain rooms. A room resides in a zone, and agents must migrate to that zone to enter the room. Zones will be explained in detail in Chapter 3.

When an agent follows a hyperlink, it is migrated to the room that the hyperlink points to. The middleware system infers the zone of the room that a hyperlink points to, locates and

[6] The way in which an agent owner informs the world entrance daemon of a preference, or a possible payment scheme, is in the realm of providing a user interface for injecting agents, depends on the application, and is outside the scope of this thesis.

authenticates a middleware there, and ships the agent to it. By design, if an agent follows a hyperlink to a given room, it also migrates physically to a middleware ("machine") in that room's zone. Physical and logical migration takes place using a single atomic operation: if either part of physical or logical migration fails, the agent continues where it was before following the hyperlink.

2.3. Mansion middleware: components, interactions and operations

Mansion is a distributed system. Logical entities such as rooms and objects may be physically distributed, possibly replicated over multiple machines. Agents are normally not aware of this. Agents mainly interact with the system using interfaces that take location-independent identifiers as arguments, that is, agents are not immediately aware of the physical distribution of rooms or other content that they access. The Mansion middleware system provides *distribution* and *location transparency* to agents.

A central component of Mansion is a middleware process that runs on every machine that manages agents and provides an interface, the *Mansion API* using which can do useful work. The middleware uses a number of internal processes and distributed services to make a system work, such as location services and object servers. These internal components and services are collectively called the **basement.** Details of the basement are described in a later chapter.

The remainder of this chapter provides a detailed overview of the logical components of Mansion. First, this chapter introduces objects, including **room monitor object (RMO),** which essentially provides the *implementation* of a room. It explains **binding,** the method that agents use to connect to objects. Furthermore, it describes how agents migrate to another room, how they interact with a room, and how they inspect attribute sets and communicate with other agents.

All interactions in a world happen through an Application Programming Interface (API), the *Mansion API*, provided to agents as part of the Mansion system. Agents need to use this API to do any work in a world; its implementation resides in the Mansion middleware. We call the specific component of the middleware which implements the API (and which is implemented as a separate process) **Mansion middleware (MMW)**. The complete Mansion middleware system—which includes several components, such as location services, to make the system work—is simply referred to as the middleware.

The Mansion API provided by the MMW contains, among other things, calls to follow hyperlinks, to communicate with other agents, to bind to objects, and to create other agents. The Mansion API guards the conceptual integrity of the system—that is, it ensures that the logical constraints of the Mansion paradigm are enforced.

2.3.1. The room monitor object

Every room has an object called the room monitor object (RMO). The RMO is the ‘‘registry’’ of a room: it contains a list of all the objects, agents, and hyperlinks in it. The RMO is used by the middleware to find information about entities in the room. Every room has one RMO; the RMO is required for implementing a room.

When an agent enters a room, it is automatically connected to—and registered in—the room's RMO. The MMW ensures that each agent is connected to its current room's RMO, and only to the current room's RMO. This happens when an agent has successfully followed a hyperlink. The MMW ensures that all interactions of an agent with objects, agents, and hyperlinks in a room take place relative to that room. Except for interagent communication, agents can only interact with entities in a room. This way, the MMW ensures conceptual integrity.

When the MMW, internally, requires information about some entity in a room (e.g., as part of executing an API call), it obtains the required information from the RMO of this room. This then, is the essential functionality of a RMO: to store all the relevant information about all content in a room. This also involves storing attribute sets. An **RMO interface** is accessible to agents, using which they can view and potentially change attribute sets. For MMW-internal use, the RMO provides a different interface, allowing the MMW to view and change RMO content that is not directly accessible to agents.

The RMO contains a list of all objects, agents, and hyperlinks in the room, together with their attribute sets. This information is the primary means for agents to navigate in and interact with a world. How this works is described in the following sections.

2.3.2. Attribute sets

The RMO contains an attribute set (AS) for every entity in a room. An AS is a set of attribute-value pairs associated with an entity. A world designer may specify different mandatory or optional attributes (in an AS) for different entities. The creator of an entity may also specify additional attributes. An AS may be empty—if a world designer allows it—except for two fields: every AS contains a identifier for the entity relative to the room, called *EntityID,* and an AS always contains an attribute *EntityType,* which may be *Object, Hyperlink,* or *Agent.* EntityID and EntityType are automatically created by and stored in an AS by the middleware when it registers the entity in a room.

An attribute is a string of the form <*Attribute*>=<*Value*>. A default attribute for objects in most worlds is a *name* attribute, where the value is a free-form string which can be filled in by the object creator (typically, the room administrator) at the time that the object is created. Similarly, a hyperlink may have a *name* attribute. Agents can fill in (modify) their AS after entering the room.

Attribute sets are filled in at the time that an entity is registered in a room. An AS may be modified later by its owner (an object's or hyperlink's creator, or an agent). Agents can change their own attribute sets: a conceivable example of where this is useful is when an agent advertises its ''mood'' using an attribute in a gaming world. Agents can access ASes or change their own (but only their own) AS using a call on the RMO interface.

A world designer may determine certain mandatory or optional attributes that need to be in an AS for each entity type. What (mandatory or optional) attributes exist depends on the application and is defined per world (see Sec. 3.8.1). For example, a world containing multimedia files may have attributes that describe (digital) media formats; a library world may contain a *topic* attribute, *author* and *title* attributes, and/or an attribute containing a list of keywords. Hyperlinks that point to a given room may also contain a *topic* attribute.

An example attribute set of an object may look as follows:

```
EntityType=Object
EntityID=2
ObjectType=MultiFileContainer
ObjectName=MRI-scans-nii
ScanType=Head
ScannerType=3TeslaMRI
ScannerType=Philips-Achieva-3.0T-XSeries-Q-dual-16-ch
ScanResolution=0.3mm
ImageFormat=nii.gz
```

A useful attribute to have for objects is the *ObjectType* attribute. An object type is essentially the name for a particular interface; this name is defined by the object creator (see Sec. 5.2.3). The *MultiFileContainer* object shown above is a standard objects provided by Mansion. Currently, Mansion supports two types of objects: a *FileContainer* object which contains a single file, and a *MultiFileContainer* which may contain multiple files stored in a user-defined directory structure. The above two objects have different interfaces, that is, the *FileContainer* object has no method for listing the files in the container.

An ObjectType attribute is relevant if a world supports more than one type of object, since agents need to know with what kind of object they will interact before connecting to the object. If considered useful, hyperlinks and agents may also have types.

To find interesting content in a room, agents look for information in attribute sets. The RMO interface provides a method for obtaining the attribute set of any entity in the room. The RMO can also be queried for a list of entities that match a template attribute set specified by the agent. Some attributes relevant to searching MRI scans (chapter 10) are shown in the attribute set example above.

Attribute sets have similarities to semantic annotation techniques such as used in the semantic Web [120, 35]. In fact, attribute sets are equivalent to RDF / XML annotations in what they can express; the latter are sets of XML entity-attribute-value triplets, whereas attribute sets are attribute-value pairs associated with an entity. If entities such as objects or

hyperlinks are annotated sufficiently precise, and if agents know what to look for in a given world, then agents should be able to navigate through a world efficiently, to find content that matches their requirements.

2.3.3. Event notification

In addition to a mechanism for searching or ''polling'' attribute sets, the RMO provides an event notification mechanism based on attribute sets. Using this mechanism, agents can be notified of changes to attribute sets automatically, so they do not have to poll the RMO repeatedly (which can be inefficient and allow agents to miss changes to an AS).

Using a call on the RMO, an agent can request to be notified when an attribute set in the RMO matches a **template attribute set,** specified by the agent. Event notification is currently based on string matching: if an AS matches all the attribute-value strings specified in the template AS, its *EntityID* is placed in the set of matching entities returned to the caller. In contrast to the RMO method that is used for regular search of the RMO's AS-es, the event notification call is a *blocking* call. When the call *wait_for_event* is invoked, the RMO blocks until an attribute set matches the template attribute set. Internally, the RMO simply evaluates all AS-es. If no AS matches, the call blocks. Whenever a change to an AS is made, all pending event notification requests (calls) are re-evaluated. If a match occurs, the corresponding event notification call is unblocked. This mechanism is called a *continuation*.

Event notification works only while an agent is in the room where the call is invoked. As soon as an agent migrates to another room, its outstanding event notification requests will be cancelled: an agent that is not in a room may not interact with objects in the room. If an agent needs to move on while keeping track of changes to attribute sets, it can either clone itself before moving, or having its child agent moving onward, keeping in touch with it using interagent communication. Or it can ask an agent provided by the room owner (if so provided) to monitor the room's attribute sets on its behalf, and to communicate any notifications to it by means of interagent communication. In short, the basic mechanism for event notification is there—it saves resources by avoiding repeated polling of an RMO, and the calling agent process can sleep while its event notification call blocks—but it restricted within a room. If the room restriction is inconvenient, agent applications can implement a solution on top of it, for example using interagent communication.

2.3.4. Following a hyperlink

An agent can follow a hyperlink by invoking a call on the Mansion API. The call takes an *EntityID* as an argument; this EntityID is found with the AS of a hyperlink as described above. The result is that an agent is migrated to the target room of the hyperlink, if the link is valid and the target room (and a Mansion middleware process to host the agent) is available.

Following a hyperlink is an atomic operation, implemented by the middleware, which moves the agent from its current room to the target room. If something fails during the migration process, the invoking agent process will be resumed at the place in its execution where it left off, with an error code returned from the call for following a hyperlink. A successful migration implies that the agent's attribute set is unregistered from the current RMO and registered in the target room's RMO; the agent's process is killed at the location where it invoked the migration call, and started anew in the context of the target room. The middleware takes care of all the steps required.

Due to its design decision to support agents programmed in legacy programming languages (including binary C agents) on heterogenous infrastructures, Mansion supports only weak migration. Strong migration (Chapter 1) is a challenge to implement in heterogeneous distributed infrastructures consisting of commodity off-the-shelf (COTS) hardware, operating systems and software. This is even more so the case when agents consist of multiple processes, including legacy binary programs or programming languages, which may be executed by the agent as a script—as they may in Mansion. Support for legacy programs increases the ease of programming agents by allowing reuse of existing tools—for example, image segmentation/analysis programs—for specific tasks.

An agent is completely restarted from its program's main entry point if following a hyperlink is successful. An agent restarts with every succesful migration to a new room. This also applies to the case where an agent does not strictly need to physically migrate to another machine, because the target room is accessible on the same machine. Restarting an agent in all cases keeps the model and behaviour clear and consistent to programmers. For example, agents usually publish their attribute set as their first task after being started.

Besides *follow_hyperlink*, which takes the room-relative *EntityID* of a hyperlink as an argument, jumping may be allowed. The *jump* primitive takes a world-wide unique *RoomID* as an argument. The world designer determines whether the *jump* calls is enabled using a configuration option in the world design document. If the jump primitive is enabled, an agent can invoke the *jump* call on the Mansion API to move to a specific room—even if the agent's current room does not contain a hyperlink to that room. A RoomID is a globally unique, location-independent name of a room; it is the same identifier that is stored internally in a RMO with a hyperlink. The RoomID of a target room can be queried using the RMO interface. A room's own RoomID can be found by querying the current room's *self* EntityID *0*. Note that querying a room's RoomID is useful in all worlds, even if jumping is not allowed. This way, an agent can keep track of visited rooms in a list to prevent going in cycles.

As an example of when jumping is useful, consider an large-scale world in which agents search for information. An agent may be in a room that contains multiple hyperlinks to different rooms, and keep a list of target rooms to visit later. An agent can then later go to one of these rooms, even from a completely different room. This is particularly relevant to avoid traversing a set of rooms back to a room from which a hyperlink to a particular other room goes; migration may be expensive so it may be considered useful to avoid this for scalability and efficiency reasons, as will be described later.

The implications and tradeoffs of allowing jump for navigation and controlling worlds will be discussed in Sec. 2.3.12; one aspect is to constrain migration to rooms where an agent is (logically) allowed to go, given world-defined constraints on a world's (hyperlink) topology and thus agent migration.

2.3.5. Agent container

Because Mansion uses weak migration, agents need to have some way to store data before, and to retrieve data after, migration. Agents store and retrieve data explicitly in a private data storage system associated with each agent. This data storage abstraction is contained in the middleware. Because Mansion does not determine the programming languages in which agents can be programmed in advance, a platform and language-independent data storage system called an **agent container (AC)** has been designed. This design is described in detail in Sec. 4.4.1.

Every agent has an AC. An AC is essentially a small, private file system associated with an agent. The AC is fully accessible to the MMW process which manages the agent. Part of the AC can be used to store administrative information needed by the MMW, and part of it can be accessed by an agent to store, retrieve, or remove data. An agent's owner can, upon collection of an agent, see all information in an AC.

Data is stored in **segments**, essentially files, which can contain binary data. Agents use POSIX-like methods for creating, opening, reading, writing, and deleting data segments in their AC. These methods are provided to agents as part of the Mansion API. Agents use their AC to store information, and to retrieve this information at any later time. Some data may be stored permanently (i.e., it can be marked as immutable) so that it cannot be removed at a later time, while other data may be modified or removed later.

In addition to providing a storage mechanism for agents, the middleware may (temporarily) store information in an AC for internal use. An example is where a MMW process, upon migration of an agent, stores data received through one of the agent's communication channels, which has not yet been read by the agent (Sec. 8.2.13). This data is stored in a temporary segment in the AC that is not visible to the agent.

As a means of protection, certain data segments can be marked as persistent in the AC. A **persistent segment** is protected against tampering by a later host on the agent's itinerary. If an agent's AC is retrieved by the agent's owner, its integrity can be verified—segments may not be modified or removed after being marked persistent. Integrity of ACs is also verified when the agent migrates (Sec. 8.3.1).

If required, data can be encrypted by an agent before storing it in the AC (e.g., using the agent owner's public key, to protect it from being read by anyone except the agent's owner). This is however handled at the application layer outside the scope of Mansion.

2.3.6. Jailing

To protect a system from potentially harmful mobile agents, Mansion uses a jailer (Chapter 6) to confine agent programs. The jailer uses system call interceptions to prevent agents from writing files outside a designated scratch directory, and from making connections to the outside world except to make Mansion API calls. A jailed process can be suspended, resumed or killed. Jailing is an effective way to ensure that:

- An agent can interact only with a local directory and the Mansion API to communicate with other agents, to follow hyperlinks, and to interact with objects. This ensures that agents are constrained to Mansion operations, which can be governed by Mansion's middleware system.
- System time and other resources can be protected. The jailing system can enforce constraints such as a maximum number of threads, maximum disk or memory usage, etc. By configuring resource management options (appendix 3), resource exhaustion attacks on systems that host agents can be prevented. agents can be prevented consuming too many resources, compared to other processes on the system.

Different agents are started up in different jails by the MMW. The jailer is agent-agnostic; agents are just regular processes to it. Agents can create child processes (modulo limits imposed by the middleware). Agents can be implemented in multiple languages: the jailer supports binary programs, scripts or interpreted programs. An agent's code (or script) is found in its AC, copied to a scratch directory, and started from there.

2.3.7. Cloning

An agent can clone itself. A **clone** is created by making a copy of the agent's original AC (so, without segments of the parent that were added after the parent's injection), and injecting this copy so that it is started as a new agent in the agent's current room. The clone has a new *AgentID* which is obtained by the parent. A world designer (or a world entrance daemon) may impose a limit on the number of agents that may be cloned, typically a maximum number of agents per agent owner. This maximum depends on implementation. For example, a system for injecting agents may make use of a Web page for registering agent owners which takes payment. The payment scheme may determine the number of agents that an given user may run at the same time.

2.3.8. Interagent communication

Agents can communicate with each other using reliable, ordered, and secure (integrity and

confidentiality-protected) communication channels. These channels have semantics similar to TCP. Channels are bidirectional. Agents are provided with a BSD socket-like interface. Each agent has a communication endpoint to which other agents can connect. After the connection has been accepted, can agents send and receive messages (variable-length byte streams) over the connection.

Agents connect to other agents using unique, location-independent **agent identifiers (AgentIDs)**. An AgentID is assigned to an agent when it enters a world. An agent can also connect to another agent in its current room using the other agent's (room-relative) EntityID. If useful, agents can then pass their unique, worldwide AgentIDs to each other after that. Agents can also announce their AgentID in their attribute set. A disadvantage of the latter approach is that agents may get spammed by other agents.

Communication channels are reliable. Agents use a socket-like interface to send and receive data over their communication channels. Connections are migration-transparent, except that a transient error may occur when data is sent while the agent at the other end of the channel migrates. Then the sending agent should retry. Data that is successfully sent will arrive reliably and in-order with respect to messages sent earlier or later. Barring persistent underlying network errors or failure in the MMW processes that host the agents, connections are maintained. The implementation of interagent-communication channels will be discussed in Sec. 8.2.13; the API calls are described in chapter 9.

2.3.9. Objects

Agents can interact with objects using synchronous method invocations. Conceptually, an object in Mansion is similar to a C++ or Java object. Objects are passive. Objects do not have their own thread of control: code execution starts when an invocation is made, and ends when the invocation returns. Objects cannot invoke other objects.

Each object has a specific *type*, which is a synonym (name) for the object's interface. Each interface consists of a set of methods. Methods are named and consist of a number of arguments and a return value, which is always a signed integer.

Object interfaces are defined using the Mansion **interface definition language (IDL)**. The arguments of object methods may be **in, out,** or **inout** variables. An in variable is passed to the object. the invocation. An out variable is filled in by the object and passed back to the invoking agent. An inout variable is passed to the object and is writeable by the object, it may be changed on return. The return value of a method invocation is typically a positive number indicating success, or a negative number indicating failure (e.g., an error code). Positive return values (or 0) are method-specific, for example, the number of bytes read. Negative values always indicate failure. Some negative error codes are reserved and defined by Mansion to indicate failure in the middleware or in the mechanism to invoke an object.

Objects in Mansion are *distributed* objects. This means they may reside on another machine than where the agent runs. The machine where the object resides is trusted (and,

likely, owned) by the room owner, and the physical location of the object and its distribution is hidden (invisible) to agents. Invocations of an agent on an object are routed to the object transparently, by the Mansion middleware.

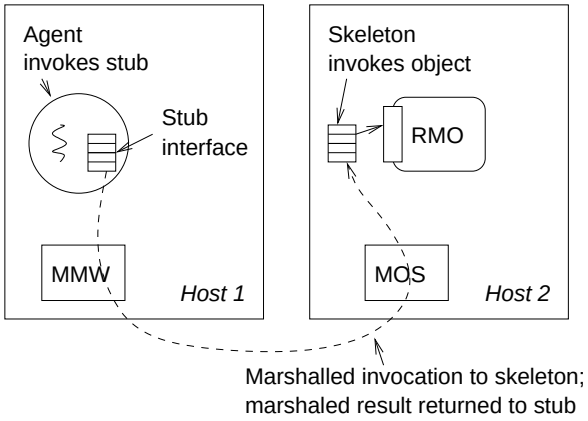

Fig. 1. An agent invoking a method on a stub object, which marshalls the request and forwards it to a skeleton, which unmarshalls the request and invokes the object; results take the same route back. Note that in Mansion, invocations are routed through the Mansion middleware (MMW) to the MOS where the object resides; this is part of binding and transparent to the agent and the object's stub/skeleton code (see text). Details on the MOS are described in chapter 8.

Objects reside in a middleware component called the **Mansion object server (MOS)**. An agent has a stub interface linked into its address space with the same methods as the remote object. The agent invokes an object by invoking methods on the stub interface, which marshalls the *in* and *inout* arguments and passes them to the object server where the object was created. In the object server, a skeleton unmarshalls the arguments and invokes the object's method (if allowed), and it marshalls and returns the results (inout and out arguments and the return value) back to the invoking object. The stub interface unmarshalls the results and returns them to the invoking agent as if the object was invoked locally (Fig. 1). Note the similarity to RPC [27]. The remote object invocation mechanism allows for transparent extension of the machines that host agents, by allowing agents to transparently invoke objects that reside in an object server on a different machine, with each machine being controlled by the room owner. Chapter 3 will explain the trust model and mechanism that ensures that room owners can manage the set of hosts that a room runs on and is available from in a secure way.

Stub and skeleton code are generated by the Mansion IDL compiler. The Mansion IDL compiler takes a Mansion interface definition as input, and generates C and C++ stubs and a C++ skeleton. The skeleton is used as the basis for implementing a Mansion object. It is straightforward to generate stubs for different languages, like Java or Python: since a stub's main function is to marshall and unmarshall arguments, generating a stub for a given language is not very complex. Skeletons can also be generated for different languages. An interface definition resembles a C++ object's interface, except that arguments only take

simple C-like data types or structs; complex arguments like objects are not allowed. The data types that can be defined in the IDL are simple on purpose for interoperability.

A world may contain different object types. Currently, *RMO*, *FileContainer*, and *MFC* (for *MultiFileContainer)* object types are defined. Allowed types for a given world are included in the world design document (WDD). Only the RMO object need not be defined in the WDD as it is a standard object predefined for all Mansion worlds, which cannot be changed (except for its content, attribute sets, which are defined in the WDD, but changes to which do not change the RMO interface).

The RMO is the only standard (mandatory) object type in all Mansion worlds. The RMO is, as far as its IDL and implementation are concerned, a regular object. It has methods for obtaining attribute sets, which the agent can invoke just as it can invoke any object. The RMO is mandatory because it is crucial to the functioning of Mansion.

There currently does not exist an *ObjectType* naming convention or name conflict resolution protocol; object types are simple strings which are known to all users in a world, as they are listed in the world design document. Naming conflicts seem unlikely; however, if ObjectType naming conflicts ever become an issue, these may be resolved using to-be defined object naming conventions or a (global) Mansion object type registry.

Mansion does not enforce any particular structure or content upon the objects in a world. Objects may store files, but they may as well store tuples or something else; obviously, objects with different purposes will have different methods. The only thing that every Mansion object has is an IDL and an ObjectType. Also, all Mansion objects must implement a standard set of access control related methods (chapter 7), that are automatically generated by the IDL (skeleton) compiler. These methods combined are called *MansionObject*. Conceptually, *MansionObject* is inherited by all objects (including the RMO) in a world. Objects are currently implemented as C++ objects, but they should also be implementable using Python or Java, or even C. The object invocation model, stub/skeleton compiler, and interface definition language take a minimal approach to marshalling invocations and data representation. This way, Mansion objects and agents (which use object stubs) can be implemented in any programming language.

The main reason for calling Mansion objects ‘‘objects’’ and not, say, ‘‘RPC services’’ (which in terms of implementation they are very close to) is that an object server may host several instances an object type, each holding its own state.

2.3.10. Binding

Before an agent can invoke a given object, it must instantiate an interface in its address space. The *ObjectType* (listed in an attribute in the RMO) indicates to an agent which type of interface, i.e., which stub, to instantiate. Next, this interface must be connected to the relevant (remote) object instance. The latter connection is coordinated by the MMW. The process of instantiating an object interface and connecting it to an object instance is called binding.

Binding is the process where an agent connects to a remote object. Part of this is setting up the local object interface (of the right *type*) in the agent's address space. The binding mechanism is partly implemented by the agent's runtime system, and partly by the MMW. To bind to an object, two things are required:

- The object's communication endpoint should be located and connected to, typically using a connection routed through the MMW[7]; and
- The interface established in the agent's address space should match the signature (ObjectType) of the remote object instance that is being connected to.

The first aspect is taken care of by the MMW, which internally uses a world-specific object location service to locate a *contact point* where a given object can be reached (chapter 3); as input for this lookup, an agent specifies the object's EntityID. Using the EntityID, the MMW looks up the RoomID as registered in the RMO of the agent's current room, which is subsequently used to resolve the object's physical contact point for binding. The connection is routed through the MMW process for security reasons. This is described in more detail later.

The second aspect relates to the issue of object typing, as discussed above. An agent (or its runtime system) must instantiate a local interface that corresponds to the object's type. In Mansion, this comes down to (dynamically) instantiating an object-specific stub which was generated by the Mansion IDL compiler. The ObjectType indicates which stub to instantiate, and what methods the agent can use to invoke the object.

Stubs are language bindings of an object interface, that is, they are generated by the IDL compiler, specifically for an agent's programming language. If an agent is written in C, it can be statically linked to a library containing a C language binding (stub) of the object's interface, which can be invoked after binding to the object. In a C++ agent, dynamic instantiation of a C++ class object can be used by the runtime system to implement the *bind* operation.

The end result of binding is that an agent has an instantiated interface in its address space. If invoked, the method is invoked on the appropriate remote object instance.

2.3.11. Confined rooms

A confined room is a special room, designed for security reasons (Section 1.4.7). An agent in a confined room cannot communicate with entities outside the room or store anything in its AC. A **confined agent** cannot take anything with it when it leaves the room. Confined agents can safely search confidential or other sensitive information in a confined room, while the room owner can be certain that no information leaks out while the agent searches information. A special **guardian agent** provided by the room's owner is used to export selected information out of the room in a controlled way.

[7] An exception is an authorised administrative program which binds directly to an object in an object server.

Implementing confinement is facilitated by Mansion's weak migration model. Whenever an agent migrates to another room, it is killed and restarted in the context of the new room. This happens every time an agent migrates, so the model is consistent. As a result of weak migration, agents are automatically brain-wiped when moving to a new room. No memory or thread state is retained during migration. It is sufficient to ensure that agents in a confined room cannot communicate with agents outside this room or write anything to their AC to fulfill the requirements for confinement. Agents can communicate with each other while in the room; as none of them remembers anything afterwards, this is not a problem—and this way they can collaborate, if useful.

The MMW takes care of ensuring that confined agents cannot communicate with agents outside the agent's current room or write to their AC; it simply disables the API calls needed to do so.

Export of information takes place either by explicitly communicating with a *Guardian Agent (GA)* provided by the room owner. The GA can be a regular agent found using its attribute set. A simpler way to obtain the same result is to leave a special *export file* in a private directory that every agent has access to, before leaving the room. This file can be picked up by a script or a program provided by the room owner. This program takes the role of the GA. This is how the GA is currently implemented.

Using the export file, an agent can provide selected information such as the names of interesting files found in a *MultiFileContainer* object in the room to the GA, which can check and filter this information. The GA can ensure that a file is stored in the Agent's agent container which contains contact information—for example, an email address with a unique transaction identifier, a website URL where the agent owner can buy information, or the AgentID of the GA from which information can be obtained. The agent or its owner can obtain the allowed (filtered) information using this information as provided by the owner of the confined room or the GA. If the confined room has a real GA, the agent can contact the GA using the GA's AgentID to obtain information; alternatively, the room owner may contact the agent's owner instead—a "do not call us, we call you" option.

The semantics of the GA interaction is application specific. The approach depends on the purpose of confinement and on the sensitivity of the information in the confined room. An e-commerce application may be less sensitive than a medical application; selected music files may be readily available through a link placed in the agent's AC and bought online by the agent's owner after the agent returned; in an application for searching medical research data, contact may only be made after the agent's identity and the relevance of the proposal for research (as indicated in the export file) is evaluated by the hospital board—only if sufficient patients matched the agent's search in the first place—after which the hospital may contact the researcher and negotiate a contract before any data is exchanged. The confinement principle allows for many usage scenarios. Examples are discussed in chapters 9 and 10.

2.3.12. About jumping, and some words about world structure

This section discusses some aspects related to world topology, hyperlink constraints, and whether or not to allow the *jump* primitive. Sec. 2.3.4 described the *follow_hyperlink* primitive that is normally used for navigation in a world, which takes a room-relative *EntityID* as an argument, and the *jump* primitive which allows jumping to an arbitrary room given a world-wide unique RoomID. Whether jumping is allowed is an application-specific world design parameter, defined in the world design document (Sec. 3.8.1).

It is instructive to discuss the various tradeoffs underlying the decision to allow jumping in some worlds, and not in others.

If the jump primitive is enabled, an agent can invoke the *jump* call on the Mansion API to move to a specific room—even if the agent's current room does not contain a hyperlink to that room. Without jump, an agent can only move to another room by following a hyperlink.

The most important considerations to (optionally) allow the jump primitive are related to the cost of cloning and migration, in particular to avoid that an agent has to go back to a room by back-tracking through a set of already visited rooms. With very large worlds that do not allow agents to find information efficiently without following a lot of hyperlinks, jumping may be a useful world design option.

As an example of when jumping is useful, consider a large-scale world in which agents search for information. An agent may be in a room that contains multiple hyperlinks to different rooms. Before following one of the hyperlinks, an agent can check whether it has been in the target room before by checking a ''visited rooms'' list it keeps. The agent also keeps a second list containing all target rooms it still needs to visit. It can construct this list by collecting the RoomIDs for all hyperlinks in rooms it visits[8]. As an (exhaustive) search strategy, an agent can use its list of unvisited rooms to jump to once it is done searching a set of rooms behind a hyperlink. This way, an agent can avoid visiting rooms multiple times, as could happen when back-tracking to an earlier room to follow a different hyperlink from there. This saves the agent double work and time—as will be shown later, migration can be rather expensive. Alternatively, cooperating agents—possibly cloned agents, Sec. 2.3.7—can search a world together by informing each other of which rooms they visited or still have to visit; depending on the number of agents that may be cloned and the structure of the world, jumping may be convenient.

When a world designer does not allow jumping, it is straightforward to force agents to follow hyperlinks to rooms in a certain order, assuming that room owners do not cheat and adhere to the rules set by the world designer. Hyperlinks allow world and room designers to enforce a structure or ordering on the hyperlinks that an agent can follow, either for the world

[8] The list of rooms that still need to be visited, may be manipulated by a malicious host on the agent's itinerary, for example to avoid that agents visit the room of a competitor. However, by making persistent copies of the room lists regularly, it becomes possible for the agent's owner to check for any malicious modifications of the (nonpersistent) lists after the agent returned, using an automated audit procedure. This way, a misbehaving hop on the agent's itinerary can be exposed. As a consequence, the agent's owner can inform the world owner who can take action—possibly removing it from the world. More information on audit trails is given in a later chapter.

as a whole or within a set of rooms, for example in a zone. For example, an agent may first have to enter an anteroom before going onward into, say, a music store. For a gaming world organised as a dungeon, enforcing structure on the world (i.e., to force agents to go through a sequence of rooms) is also clearly useful. The use of hyperlinks to structure worlds was the reason to not support jumping originally.

Control over iteneraries is, however, not lost completely when using jumping, even though itineraries may not be enforceable in as fine-grained a way as when using hyperlinks to structure migration between rooms. Actually, control at another level is required also when using hyperlinks to structure worlds, since with large-scale worlds it becomes very hard if not impossible verify adherence to fine-grained hyperlink constraint policies. Thus, a world designer can express constraints in the world design document that describe what zones may link to what other zones. These policies are described in the following section.

Constraints on migration at the zone level can be enforced by the middleware, irrespective of whether a hyperlink was followed or a jump call was made; at the time an agent migrates, both sending and receiving middleware processes authenticate each other as a member of a zone in the world, and at that time can check whether migration is allowed given the world's topology constraints; further, auditing of agent itineraries can take place in the central agent location services. Details of the mechanisms required will be explained in chapter 3 and 8. The following sections describe some details of migration and topology constraints.

2.3.13. Physical migration

Agents interact with rooms by inspecting content in an RMO and interacting with objects (or agents) within a room. Following a hyperlink brings an agent to another room. A key question at design time was whether physical migration is mandatory or optional. Mansion could provide an *option* to agents by which they could choose to access a room (i.e., its RMO) remotely using a form of RPC, instead of forcing agents to migrate there.

We chose for physical migration instead of supporting a client-server approach. The reason for this decision is that we aim to provide a clear, unambiguous model regarding migration, which distinguishes itself from the Web approach where content is downloaded to the machine where the client resides. Section 1.3 motivated the design decision to force use of *mobile* agents in all cases; we make the discussion more specific here, now that we discussed Mansion's conceptual model.

The client-server approach of the Web has shown to be extremely successful. Had Mansion been implemented using a client-server approach, a client agent could interact remotely with Mansion objects such as the RMO using a form of RPC, similar to a Web browser fetching Web pages or putting content on them. In this case, Mansion would not have many benefits over the Web. A hybrid model, where agents would be aware of distribution aspects and make conscious migration decisions instead of being always forced to migrate could also have been possible. However, we wanted to have a simple, consistent programming model

that emphasizes the benefits of using mobile agents. For the purpose of consistency, we decided that agents migrate *always*—even when following a hyperlink to a room within a zone, at which time migration may not be strictly needed.

The reason Mansion does not use a client-server approach, is that it undermines some of the strengths of using mobile agents; a hybrid model leads to unclear semantics. Without enforcing migration, it would become difficult to meet the requirements and explore the use cases outlined in chapter 1, such as searching sensitive information *on-site*. Also, consistency of the model is difficult to maintain: were objects including the RMO accessible from any machine, how easy would it be to guard the design decision that agents can visit only one room at a time? The Mansion source code could be modified easily such that agents could connect to multiple RMOs/rooms at the same time. It would thus be difficult to impose constraints or policies on, for example, the way that a client navigates in Mansion. Thus, the most important reason for enforcing agent mobility is that it allows the system (middleware) to maintain a consistent view of how a world looks, including structure and security aspects.

The idea of Mansion is that a world's structure (e.g., a predefined itinerary of rooms) can—if needed—be imposed on agents. Mansion allows for defining world-specific constraints on hyperlink layout, to force agents to take a certain path from a world entry room to some other room. Also, without physical migration, implementing *confined rooms* that allow data owners to control export of information, while allowing agents to freely search content while in the room, would be infeasible.

2.3.14. Hyperlink topology and navigation

Hyperlink constraints are important to structure a world. They are defined in the world design document. The granularity of hyperlink constraints may differ from world to world, as a world's hyperlink layout differs from world to world.

If defined, a hyperlink constraint provides a policy on what zones may point to what other zones. At the world level, what rooms can be pointed to cannot be controlled; such detail would become too difficult to manage and managing world topologies would not scale. Rooms are part of zones, and zones can agree between them what rooms may point to what zones. Zones *may* define certain zone entry rooms as the official zone entrance rooms, and refuse agents from outside the zone to enter a non-entry room. Zone owners may also have agents enter specific rooms depending where they came from. This is however not visible at the world level or in the world design document.

If defined, hyperlink constraints apply to jumping as well as to following hyperlinks. This way, a world's topology can be controlled consistently. For example, in a world that allows jumping, an agent may store a RoomID in its AC to jump to later. Jumping to this room may not be allowed at a later time, if a hyperlink constraint forbids it from where the agent is at that time.

A world's structure has obvious consequences for search—for example, if a world has no back-links, or if its hyperlinks do not form a tree-like or spanning structure, searching a world may become complicated, especially if the world is large. It is evident that a world designer must communicate a world's topology to agents or to agents and their programmers.

The constraints of a world's topology may be communicated informally using instructions to owners of rooms in a world, or more formally through hyperlink constraints defined in the WDD. Another approach is to communicate topology information within a world.

Hyperlinks are unidirectional, and Mansion does not impose a particular way of using hyperlinks in worlds. A Mansion-provided utility program for creating rooms by default creates a back-link from a new room to the current room, where the back-link can be distinguished form other hyperlinks by its name. However, this is not mandatory and another tool may do things differently.

Early in the design of Mansion, we defined an *Attic* for the purpose of providing services to agents, such as yellow page services (listing company rooms), white pages (listing AgentIDs) and topology information. The **attic** was a special room that could contain **service objects** (invokeable in the same way as regular objects) or (non migrating) **service agents**. A world could have one or more attics and would be registered in a room's RMO. The goal of an attic was to allow a world owner to provide some common services for a set of rooms.

The attic concept was not implemented because other mechanisms exist that can be used to the same purpose. Attic-like services can be deployed in a world entrance room, where agents can obtain information about the layout of a world from objects or other agents, prior to following hyperlinks into the world. A service agent in a WER can be contacted for information at any time. Furthermore, hyperlink constraints and other important information about a world can be communicated to users and agent programmers externally. Hyperlink constraints at the zone level are part of the world design document which can be obtained by agent developers (see Sec. 3.3).

An attic has the advantage that it may be accessed at any time, irrespective of what room an agent currently resides in. An attic could contain a roadmap listing important rooms with their RoomIDs—for example, rooms that are entry points to ''sections'' on a given topic—useful for navigation. However, world-wide availability of services such as a roadmap service can also be achieved by service agents in a WER; for the prototype world described in this thesis (chapter 10), there was no need for an attic.

A way to help agents navigate large worlds is to construct a database that is equivalent to a search engine in a world entrance room. Such a service may contain the RoomIDs of rooms matching a query. Alternatively, for a world without a *jump* primitive, such a service may return a set of possible ''paths'' (consisting of a sequence of hyperlinks to follow) to take to reach a given room matching a query.

Agents in a world may map the (dynamic) hyperlink structure and possibly content of a world on-the-fly and help other agents navigate a world, by providing agents with information about interesting routes to follow, either by placing information in a search engine equivalent, or by exchanging information by communicating with other agents.

2.4. Putting things together

Figure 2 shows an overview of the main (logical) components of an example world.

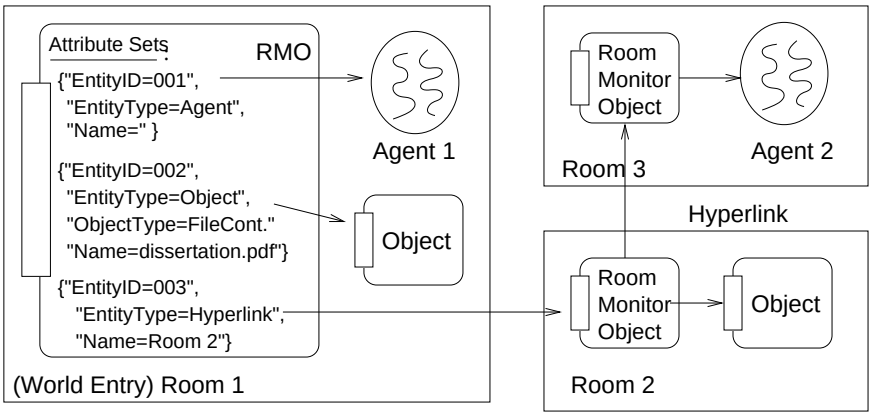

Fig. 2. Overview of a Mansion world with three rooms. See text for explanation.

The world in Fig. 2 contains three rooms. Each contains, by definition, a room monitor object (RMO). The RMO contains information about entities in the room, including attribute sets. Visible are objects, agents, and hyperlinks to other rooms. This world uses an attribute set that contains only a Name attribute for all entity types. The world designer specifies whether attributes are mandatory or not. In this case, the Name attribute is not mandatory, entities may leave this attribute empty (e.g., agent).

The world has a simple hyperlink layout. A world entry room (entered when an agent is injected into a world), room 1, contains a hyperlink to room 2. Room 2 contains a hyperlink to room 3. There are no back-links in this world; thus, agents must follow a path from room 1 to room 3, unless they exit using an API call (the result is that the agent is transported to the morgue—not shown).

Arrows indicate show which entity an attribute set (AS) refers to. Rounded rectangles are objects, the rectangle to the left visually representing its interface. Ovals indicate agents containing threads (agents are processes). Internal information in the RMO, used by the MMW internally to locate entities, is not shown in this figure: the RMO contains information which is invisible to agents that allows the MMW to bind to an object (Sec. 2.3.10).

Only ASes in the RMO of room 1 are shown. The AS of EntityID 3 contains a description of the hyperlink to room 2. Different EntityTypes can be seen for different attribute sets. Agent 1 only uses the standard EntityID and EntityType attributes. An Object has an attribute Name, and a hyperlink has an attribute Name which describes the target room (this attribute could also have been called *TargetRoom*).

One object has an *ObjectType* ‘‘FileContainer’’ with *Name* attribute ‘‘dissertation.pdf.’’ From these attributes, an agent can derive that this object has (IDL) type FileContainer and it

contains a PDF file called dissertation.pdf. Note that a world where objects may potentially contain many file types, an attribute ‘‘FileType’’ may be useful; MIME-types[9] like strings can be used to indicate the file type, e.g., application/pdf.

2.5. Discussion

This chapter described the architectural elements and the main components of a Mansion world. Agents interact with objects in the context of the room that they are in, and can interact with other agents using migration-transparent communication channels. Agents can collect information and store it in their agent container, or send the information home over a connection to their owner. In a confined room, agents can search but they cannot export information. Weak migration ensures a consistent semantics when following a hyperlink, irrespective of whether an agent is in a confined room or a regular room.

A world design document contains basic information on hyperlink (zone) constraints, and additional information that help agents navigate can be found in a world, either using services found in the world entrance room or by contacting other agents in the world. World topologies can differ depending on application requirements, with more structured worlds likely disabling the jump primitive, while other worlds may allow agents to jump around more freely. In all cases, attribute sets are used to annotate content (rooms, agents, objects and content) so that agents can find their way. Agents migrate physically, so that they are always close to the data they inspect, leading to potential benefits in efficiency and customiseability of search, and security and controllability benefits for content owners.

The aim of Mansion is to provide a consistent model that allows for efficient search in a structured environment, and to provide a paradigm that emphasizes the benefits of using mobile agents. This chapter introduced the paradigm and its components, and describes how it helps application developers develop worlds. The next chapter describes how a mansion world can be deployed on physical infrastructure (hosts), and how Mansion aims to provide scalability and controllability properties of this environment.

.

[9] MIME (Multipurpose Internet Mail Extensions) types are defined in RFC 2045-2049

.

Chapter 3
Distribution Control and Scalability

Mansion rooms, objects and content can be provided by different users, and it can be hosted on different servers (by different owners) all over the world. Mansion should scale in view of the overall number of machines, rooms, objects, and agents in a world. It also has to scale administratively and it should allow world owners to retain control over relevant aspects of their world, even at large scale.

Several important components are implemented as distributed objects, which can be transparently replicated over multiple hosts. Transparent replication was the goal of the Globe distributed object system [107], and the use of distributed objects is one of the starting points for the Mansion design[10]. Agents can access a room and its content as distributed object(s) without realising that these objects are distributed and may run on a different machine. Object distribution makes it possible to scale a room transparently with regard to the number of agents that it can handle, by adding object replicas and/or by adding machines from which (more) agents can access these objects.

Distribution transparency is a goal, and is obtained to an extent: rooms and objects can be distributed transparently to ensure they can cope with increasing load. However, full distribution transparency is not realistic for mobile agent systems. In particular, physical distribution aspects relate to security and *trust.* Not all underlying processes or systems, or all administrators or users of the underlying systems, trust each other equally.

Agents may not migrate to all systems as their owner may not trust all content or physical systems (or their administrators) equally. When placing (sensitive) content in a world, the content owner may only trust a few machines to host that content. In short, security and trust issues stand in the way of full location or distribution transparency, and ways are needed for administrators and users to find out where information resides physically, and to control aspects related to system and content distribution.

[10] In the end, Mansion objects were not implemented as Globe objects, but the design principles are applicable.

This chapter describes how Mansion has been designed to scale by providing location and distribution transparency when possible, while also allowing reasoning about and control over (security) relevant properties of the system.

3.1. World deployment

A way is needed to map the paradigm's logical model onto a physical model. Mansion's logical model is about rooms, objects, hyperlinks and agents. The physical model consists of administrative domains, machines, and (middleware) processes running on these machines. The mapping between the logical and the physical model is related to administrative and deployment aspects, and (implicitly or explicitly) to *trust.*

The concept of a **zone** maps the logical model onto the physical model.

A zone is a grouping of (middleware) processes under single administrative control, where each process may run on a different machine. Each zone member process must be authenticateable as a member of the zone.

Content in a world (in rooms) is placed in zones. Zones thus create the mapping between the logical world (rooms, objects, content) to the physical world (machines, processes). Zones are essential to understand how the logical world is mapped onto the physical world—that is, how the content of a world is physically distributed—and for security.

Each world consists of one or more (disjoint) zones. Every zone consists of one or more Mansion middleware processes on one or more machines. The middleware processes contain cryptographic keys using which they can be authenticated as a member of a zone. Every room is located (distributed) in one zone only. Objects in this room are located in the same zone. A zone can contain multiple rooms.

An agent has to migrate physically to a machine in a given room's zone before it can enter that room. Concretely: when an agent follows a hyperlink, the agent's MMW must select a MMW process in the target room's zone, and ship the agent there.

This chapter explains the zone concept in depth, including administrative details. Important actors in (physical) world deployment and in zone management will also be explained. The chapter concludes with an overview of the most important middleware components needed to make the Mansion system work, and describes how these are mapped onto the zones that constitute a world.

3.1.1. Administrative entities

A world has the following administrative actors.

- **World owner:** the world owner or administrator is in charge of the design and deployment of a world. Ultimately, the world owner decides on the content of a world – both in logical terms (what is the world about, what are useful descriptions of entities in the world, and what are useful hyperlink constraints?), and in physical terms: what zones may be part of the world, and under what conditions?
- **Zone owner:** the owner of a zone, which is a group of processes on a set of machines on which content such as rooms, objects, and agents can be hosted. A zone is an administrative/organisational grouping. A zone owner manages and trusts the processes hosted in the zone, and is responsible for them. Usually, a zone has a name, and possibly other properties associated with it. A world administrator can remove a zone from the world if it does not adhere to the world's requirements.
- A **room owner:** someone deploying a room and managing its content (e.g., data in objects in the room). Often, a room owner is the same principal as the zone owner, but this need not be the case; rooms can be deployed in a zone that the room owner does not have control over, much like a Web page can be hosted by a third party.
- An **object owner** is typically the same person as the owner of the room in which the object resides, but this is also not necessarily the case. Content in an object can be provided by someone else than the person managing the object.
- A **world entrance zone owner.** World entrance zones are special. A world entrance zone contains one or more world entrance rooms and a number of important *trusted* services. These services allow users to inject agents into the world, keep track of them, and store them when they exit so they can be collected by their owners. These critical services are managed by the world entrance zone owner, who needs to be trusted by the world owner, content providers, and by agent owners that make use of this zone. There may be more than one world entrance zone per world. The world entrance zone owner and the world owner may be the same person.
- **Agent owner:** the owner/user who injects an agent into the world. The agent owner is generally authenticated (identified) by a world entrance daemon in a world entrance zone, at the time of agent injection. Only the agent owner can collect an agent when it has exited, and may directly communicate with their agents.

The terms ‘‘owner’’ and ‘‘administrator’’ are often used interchangeably in this thesis. Owner and administrator may not be the same person. For example, an owner may be a multinational organisation, while an administrator is a technical expert who ensures the system work. However, in general, an owner is the responsible entity for whom an administrator works, so owner and administrator are related entities.

3.1.2. Mapping Mansion components on zones

The abstraction chosen to map a logical model on a physical model is called a **zone.** A zone is effectively a set of processes. Each room is mapped on exactly one zone. Rooms are only distributed to—and accessible from—middleware processes in this zone.

A zone is a grouping related to adminstration and security. Zones are sets of middleware processes that can be authenticated using a single *self-certifying name* (Sec. 3.2.1). A zone owner decides to run/selecting or accepting suitable processes on suitable hosts in a zone. The management of content and composition (membership) of zones is decentralized and the responsibility of the zone owner for scalability and autonomy reasons: when middleware processes and machines need to be added to a zone, this should not require world owner involvement.

A zone may consist of several different middleware processes running on different machines—from one to potentially hundreds of machines spread over the physical worlds. The zone administrator determines how these processes are distributed over a zone. A typical zone contains the following processes, that may physically run anywhere in a zone:

- One or more MMW processes for receiving, running, and shipping agents
- One or more object server processes, running one or more objects (at least one RMO, and regular objects)
- Location service processes that are used to find processes, rooms, objects in a zone.

Example zone member processes are the MMW that hosts agents and the Mansion Object Server (MOS) that contains objects. Each zone member process can run on a different machine. Every zone must have a location service node managed by the zone owner, in which zone member processes are registered. Using this service, zone member processes can be located. The location service is an object in a MOS which is not visible to agents.

All middleware processes in a zone recognise each other using mutual authentication using a mechanism explained later in this chapter. Authentication takes place when middleware processes connect to set up reliable connections between them. Except for location services and MMW processes receiving agents, zone member processes will not accept connections from outside its zone. Zone composition as well as distribution of middleware components in a zone are transparent to agents using the system.

Fig. 3 shows a number of processes in a world that consists of two zones. The figure shows two examples of distributing processes on zones. Logically, the zone contains three rooms and a number of middleware processes. At the top, the logical view is presented. Zone 2 contains one room and an object. Zone 1 contains two rooms: RMO1 in room 1 and RMO 2 in room 2. At the bottom, the physical view is presented.

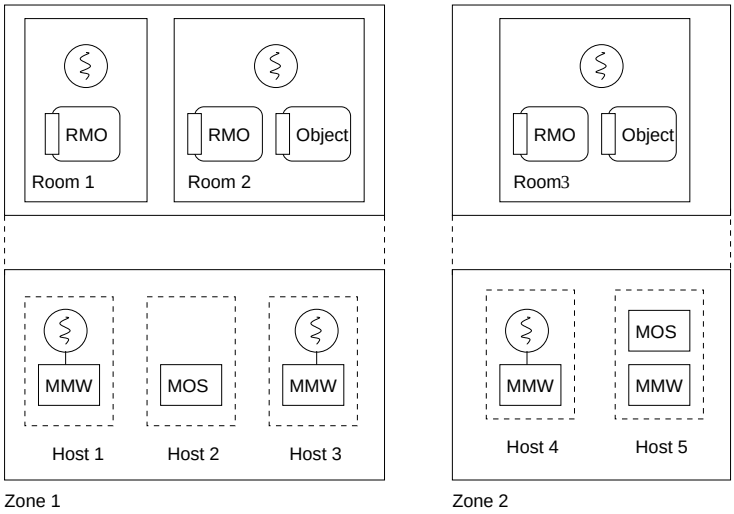

Fig. 3. Logical concepts mapped on physical concepts: three rooms mapped on two zones. Bottom shows physical concepts: MMW processes hosting agents, and the Mansion object server (MOS) hosting objects. Top the logical view: rooms (in zones) containing objects and agents.

The figure shows a few examples of distributing processes on zones. There may be one MOS hosting all objects in a zone, with multiple MMW processes accessing it (zone 2). In zone 1, a MOS runs on one machine with no MMW process. This may shield the object server from security problems that may occur, or prevent problems due heavy load generated by hosting multiple agents. Alternatively, each host could contains a MOS that holds replicas of the zone's objects. In this case, an object replica is always locally available to an agent. A load balancing strategy (implemented by returning MMW addresses from the location service in round robin order) spreads incoming agents over different MMW processes in the zone. The MMW starts the agent on its machine. The MMW transparently routes requests (invocations) by an agent to the appropriate object server(s).

3.2. Using self-certifying identifiers to name components

In Mansion, all components and entities (middleware, object server, agents, objects) are named using location-independent *handles*. These handles—essentially, character strings of up to 64 bytes—allow for registration of multiple *contact addresses* of an entity in a location service; this supports replicated objects and services.

Besides allowing for registration of contact addresses, handles in Mansion are *self-certifying*. This means that, when connecting to an entity after resolving its contact address, the entity can be authenticated using information from the handle. Interestingly, because of this approach, the location service need not be trusted as it cannot forge contact contact records without this being detected at connection time. Only by managing handles (e.g., storing them

as hyperlinks), a decentral system for finding *and* authenticating middleware components is available.

This section explains how authentication of entities using self-certifying names (zone identifiers) works, followed by other examples. After that, management of world membership, rules and components including the location service will be explained. We conclude by discussing controllability and scalability aspects of the system.

3.2.1. Zone-based authentication

Mansion components are referred to using *handles*. Handles contain an identifier of the zone that the component resides in, and its type (e.g., an object server, or an agent).

Zones are identified and authenticated using **self-certifying identifiers (ScIDs)**. A ScID is a string that allows a client process to authenticate a (remote) process.

Self-certifying identifiers were first introduced in the context of the Self-certifying File System (SFS, [65]). In this system, ScIDs were used to authenticate nonreplicated file servers. This section introduces a way to use ScIDs to identify *groups* of processes.

Implementation of ScIDs in SFS is straightforward. SFS *pathnames* contain the hash of the public key of the server on which the file is stored. The public key of the server can be contained in a self-signed certificate, where a file server has access to the corresponding private key. An authenticated key exchange protocol, combined with client-side verification of the ScID against the public key certificate obtained from the file server, is sufficient to implement authentication of self-certifying file names.

3.2.2. ScIDs for replicated services

Mansion uses an adaptation of the SFS scheme to authenticate replicated services—zone members. A *zone* is a set of processes. Each zone has a unique, self-certifying ID, called **ZoneID**. Using ZoneID, any process that is a member of this zone can be authenticated. Each zone has a self-signed certificate, called the **zone certificate**. Using the corresponding private key, the zone owner can sign **zone member certificates** for all processes in the zone. Every zone member has its own zone member key and certificate. In effect, the zone owner acts as a certificate authority for the zone. A zone certificate chain is shown in Fig. 4.

The ZoneID is the hash over the public key stored in the zone certificate. For authentication, a client retrieves this certificate together with the zone member certificate. Zone members must produce a certificate chain up to the zone owner to clients at authentication time. The current implementation uses RSA public/private key pairs, with X.509 certificates. This allows re-use of existing technology (i.e., SSL) to exchange and verify certificate chains, but the certificates normally contain no information common to X.509 certificates, since they are self-signed. Zone (member) predicates are optional in discussed in Sec. 3.9.

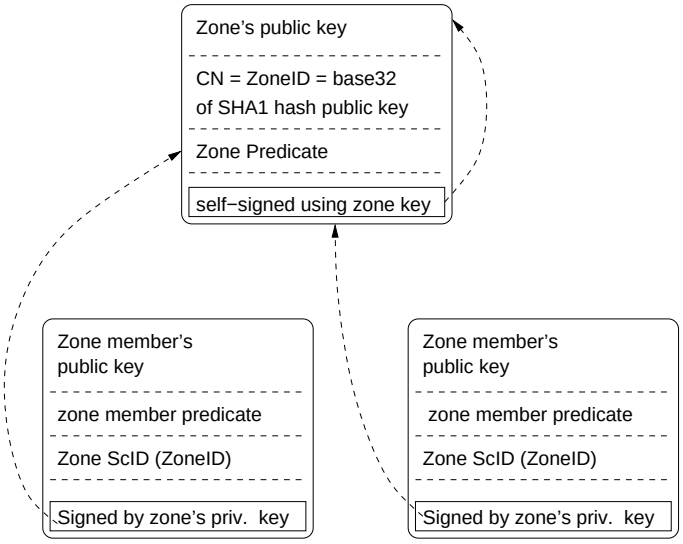

Fig. 4. Zone membership certificate chain. Shown is a zone key / certificate, which is the root of a certificate chain. The private zone key is used to sign zone membership certificates. Arrows point to the key which is used to (self)sign a certificate. Also shown are zone (membership) predicates; these are optional fields used to attach properties to zones or zone member processes. See text for details.

Mansion *base32*-encodes ScIDs so that they are human-readable; more precisely, a ScID is the base32-encoded SHA-1 hash of the public key as embedded in ASN.1 format in the public key certificate. The resulting string has a fixed, short size of 32 characters coming from the base32 alphabet. This alphabet contains no spaces or special characters, which ensures that the resulting strings can be used in, e.g., URNs. Nonencoded ScIDs are 20 bytes, corresponding to the 160-bit SHA-1 hash.

Authentication of zone member processes works as follows. First, a process authenticates a zone member process using an SSL-based authenticated key-exchange protocol (implemented using the OpenSSL library) to set up a mutually authenticated encrypted channel. As part of this protocol, the connecting process obtains the public key of the zone member process and the zone member's certificate chain. The zone member certificate is verified for correctness. Among other things, the expiration date is verified. Short expiration dates (combined with a renewal service) can be used by the zone administrator to ensure compromised zone member certificates can be invalidated quickly[11]. By hashing the key of the zone certificate, a hash value is obtained which can be compared with the ZoneID. This ZoneID is normally known in advance to the connecting party, as it is part of the entity's handle.

Various uses of (base32-encoded) ScIDs are used throughout this dissertation. ScIDs (ZoneIDs) are used as part of various ''handles'' which refer to entities such as objects or rooms. This ensures ZoneIDs are known to a process when connecting to an entity.

[11] A blacklisting approach can be used for invalidating compromised zone member certificates with a longer expiration date. Note that if one zone member process is compromised, the zone key is not compromised.

3.2.3. Other applications of ScIDs

Self-certifying identifiers are used in *handles*. Handles are location-independent names of entities such as objects, rooms, and agents. For example, a room's ZoneID is part of the RoomID of a room. The RoomID is the handle of that room's room monitor object. ScIDs can also be used to authenticate nonreplicated entities. A ScID is the hash over the entity's public key. The hash over an agent owner's key is used as an identifier for the agent's owner. This ScID is called **AgentOwnerID.**

Using a self-certifying handle, entities or processes that host an entity can be authenticated. For example, an object server that hosts a (replica of an) RMO can be authenticated using the RMO's RoomID. Similarly, a MMW process can be authenticated using the ScID in the RoomID (hyperlink), when an agent wants to follow this hyperlink.

The convenience of using ScIDs is that they provide a simple, short, fixed-size way to name entities or principals. ScIDs simplify key management because once an entity's self-certifying name is known, it is not needed to obtain its key. This collapses policy on mechanism in a simple, convenient way. (In a sense, the key management problem becomes a ScID management problem. Because of the use of hyperlinks, this problem is now implicitly solved by resp. tied to managing hyperlinks between rooms).

The fixed size of ScIDs simplifies data structures. For example, access control list (ACL) of Mansion objects contain the AgentOwnerID's of agents that are allowed access. With a PKI, in contrast, X.509 *common names (CNs)* would have to be included, in addition to requiring a PKI to bind these CNs to keys. On the other hand, ScIDs only identify keys; where X.509 certificates contain additional information about the holder of a key, such information needs to be found elsewhere if ScIDs are used. How this is dealt with in Mansion (e.g., using zone lists) will described elsewhere in this thesis.

Note that the fixed, limited size of hashes also imply that there may be collisions. In fact, there have been reports of collision search attacks on SHA-1 However, in the case of ScIDs, such attacks do not impact security. Even if some public key can be generated which generates the same SHA-1 hash (this is called a collision), it is still computationally infeasible to compute a private key from this public key—a basic assumption underlying public key cryptography (see e.g., [104]).

An important ScID is the **WorldID.** The WorldID is the ScID of a world—specifically, it is the ZoneID of the world zone. WorldID is used to authenticate a number of important services of a world, which are hosted in the basement. Each world has a special zone called the **world zone,** which contains the services that comprise the basement. These services are essential to the functioning of the world.

3.3. Mansion-internal infrastructure

Former sections described logical components that are visible to agents, and how these are

distributed over zones. This section introduces internal components of the middleware. This consists of **core information services** and the **location service**.

3.3.1. Core services of a world: the Basement

Every Mansion world needs a number of services that are managed by the world owner and which are, part of the basement and located in the world zone. The most important are:

- **Zone information service.** The zone information service contains the **zone list**. The zone list is a list of all the zones in the world, together with (optionally) information about them. The information in the zone list is vetted by the world owner. If a zone is not in the zone list, it is not part of the world. The zone information service can also be queried for information about specific zones.
- **The bootstrapper service.** The bootstrapper service contains a set of documents needed to initialise the system, including the (pre-parsed) WDD files containing attribute set definitions and object IDL definitions. The bootstrapper service is used as part of initialising middleware systems that join a world, but it also acts as a resource for agent programmers and content providers. The bootstrapper service is implemented as a *MultiFileContainer* object which contains a set of files. Its content is read-mostly, so it can be replicated with ease.
- **The world's location service root** The world location service (WLS) is the root of the world's location service; it contains pointers to (contact addresses of) zone location services, agent location service and other parts of the world's distributed location service. It also contains pointers to basement services including the bootstrapper services and core services such as the world's agent location and Morgue services. The WLS is critical to locating entities in a world. Like other services, it is implemented as a Mansion object that can be replicated. The contact addresses of the WLS are registered in a global service provided by Mansion, the Mansion Root Service (see below).

3.3.2. Delegated world services

Every world has a number of core services that are not necessarily managed by the world owner. Preferably, for scalability reasons, each world has one or a few world entrance zones that are managed by independent administrators. The world entrance zones each contain one or more **world entrance rooms**, through which agents can enter the world. World entrance zones are trusted by the world owner and, necessarily, also by agent owners that enter the world through this zone.

A world's **entrance zone (WEZ)** is marked as a special zone in the world's zone list; using this ZoneID, the entrance zone's core services can be located. These services are registered in the WLS. These are the world entrance daemon, morgue, and ALS.

The **world entrance daemon** is the system through which an agent is injected in a world. The **Agent Location Service (ALS)** keeps track of an agent's physical whereabouts; an agent's contact address is needed when another agent wants to set up a connection to an agent. Upon exit (e.g,. after an agent's time to live expired, or when it calls *exit*), an agent is migrated to the **morgue.** Here, the agent can be collected by its owner.

A combination of ALS, WED, and Morgue services must always run in the same world entrance zone. Each WEZ has to be trusted by the world owner because it is the entrance point to the world; the WEZ ensures that agents end up in a valid world entrance room, and its services must ensure that agents do not violate world design policies such as a maximum time to live, or a maximum number of child agents per agent or per account as defined in the WDD or per some (payment) scheme. It also has to be trusted by agent owners because the WEZ manages their agents while in the world. Properties of a WEZ are relevant because at world entrance the world owner and agent owner's requirements meet.

Each WEZ can implement specific constraints, rules, or controls. For example, a WEZ may be told by the world owner that agents must always be able to reach a specific set of rooms from one of its world entrance rooms. It may also be that the available rooms differ per payment scheme, with the world entrance zone administrator controlling this. For example, a platinum owner's agents may be started up in the platinum world entrance room and may be able to move to all rooms in the world, while silver agents may be start up room, from which only a limited number of rooms are reachable. There may also be ''anonymising'' WEZs which offer access through a layered, multi-organisational anonymisation scheme for privacy reasons, or which offer ''data retention free'' services. Another policy may be configured where a world entrance zone's services are required to keep audit logs for a certain period of time. Using world entrance policies, application designers gain a lot of flexibility to design policies; most of these can be enforced through the world entrance system.

3.3.3. The Agent Location Service

Agents are managed by one of the world's **agent location services (ALSes)**. An agent's ALS is located in the world entrance zone that the agent entered through. An ALS keeps track of the whereabouts and contact address of an agent, so that other agents (or the agent's owner) can contact it using the agent's unique *AgentID*. The ALS implements a home-based approach similar to that used by mobile phones. Similar to mobile telephone numbers, the AgentIDs contain sufficient information for middleware processes to locate the agent's ALS when it needs to locate the agent. The AgentID contains the ZoneID of the world entrance zone. Using this ZoneID, the *handle* of the ALS can be constructed and resolved using the WLS. Next, the AgentID can be resolved in the ALS.

3.3.4. A world's location service infrastructure

Each world has its own hierarchical location service. Contact addresses of services and processes which have to be reachable from anywhere in the world are located in the top-level part of the location service, the **World Location Service (WLS)**.

To a large extent, the **Zone Location Service (ZLS)** is responsible for providing scalability of the overall location service. Because lookups for entities local in a zone are handled by the (local) ZLS, with the lookups originating from agents running *within* the zone, most lookups will not reach the WLS. Clients cache contact addresses locally after resolving them[12]. Like all location services, multiple contact addresses can be registered with one handle, including addresses for ZLS replicas. These are returned in round-robin order or can be requested all at once.

ZoneIDs are the key to locating entities in the location service. All entities contain a ZoneID as part of the handle that they are referred by: AgentIDs, object handles, RoomIDs, middleware handles. The AgentID contains the ZoneID of the agent location service (ALS) that manages its contact records, with the ALS' handle registered in the WLS. Zone-internal handles are registered in the ZLS, except for the ZLS itself which is registered in the WLS. As an example, the ZLS can be resolved by constructing—by convention—a name *<ZoneID>_ZLS_0_0* and resolving this name in the WLS.

3.3.5. Trusting the location service

It is important to note that the middleware does not rely on trustworthiness of the location service. For a potentially very large-scale system such as Mansion, trusting a location service (and thus the MCR) would be very risky, as it would be an obvious target for attack. Further, location service should be able to scale. It should provide correct information to allow the system to work, but it should not have to do extensive security validations for all addresses that are registered with it. The core idea of using self-certifying handles is that the location server need not be trusted, as authentication takes place *end to end*, when connecting to the referred-to entity.

An important starting point for the trust model regarding usage of self-certifying handles, is that a room owner verifies the validity of a hyperlink, including the ZoneID encoded in it, before registering the link in the RMO. In addition, there is an independent list of zones, maintained by the world owner, which lists properties of each zone (Sec. 3.8.2). The room owner may receive RoomIDs in various ways, depending on the world or room's policy. This may imply extensive validation but also may simply be to trust a link when it is sent by a friend over (secure) email. Either way, key management takes place at the application level in

[12] Typically this means that the next time an entity is resolved, the address (e.g., of a MOS) returned by the resolver will be the same. In Mansion, underlying secure connections are often kept alive transparently or SSL sessions are maintained, so that connecting to the same underlying service next time is efficient. This artifact, which improves scalability, is simplified by the resolver's caching behaviour.

the form of *handle management*. Thus, the world's zone list, combined with hyperlinks and AgentIDs advertised by agents, are the basis for authenticating (target) rooms and, with that, zone member processes.

The location service simply records contact addresses. However, there remains one risk regarding the location service, and that is that it can be contain erroneous addresses, making entities unreachable. This may cause service degradation and it may even be used as a basis for a denial of service attack. Therefore, location service nodes check, for all modifications of contact records, whether the modifying party is the same as the registering party (appendix 5). Mansion object, it implements client authentication by default (chapter 7). Thus, this check is relatively straightforward and enabled by default.

3.4. Global Mansion services

Mansion has two global services for all worlds, the **Mansion root service (MRS)** and the **Mansion naming service (MNS)**. These services are available for all worlds and are currently run by the author of this dissertation.

- **The Mansion root service.** The Mansion Root Service (MRS) is the root of the Mansion location services. The contact addresses of all World Location Service (WLS) nodes of all Mansion worlds are stored in the MRS. The MRS is a replicated service. In contrast to WLS nodes, which may run on an arbitrary port, the MRS listens on a well-known port. The IP addresses of all replicas of the MRS are registered in the Internet Domain Name Service (DNS) under *mrs.mansionworlds.org.*. Using this service, a world's WLS can be located; its address can be found by looking up a name consisting of the WorldID (Sec. 3.2.3) plus extension *_WLS*. Information need only be looked up from the MRS at the time a world is bootstrapped; after that, middleware processes have cached the the WLS contact addresses.

- **The Mansion name service.** The Mansion Name Service (MNS) can be used to register human-readable names (aliases) of all worlds. WorldIDs are 32-character base32-encoded strings. The world's WorldID can be registered in the MRS so it can be looked up using the world's alias, after which the MRS can be called to bootstrap the world's location service and continue. The registration procedure is straightforward: the first person to come with a new name for a WorldID, a name that has not yet been registered, can register the name. There is no naming convention for Mansion world names. Mansion names are simple ASCII strings without spaces. If the need for more structure arises, a naming scheme like that of DNS could be considered.

3.5. Location service overview

The Mansion location service is shown in Fig. 5.

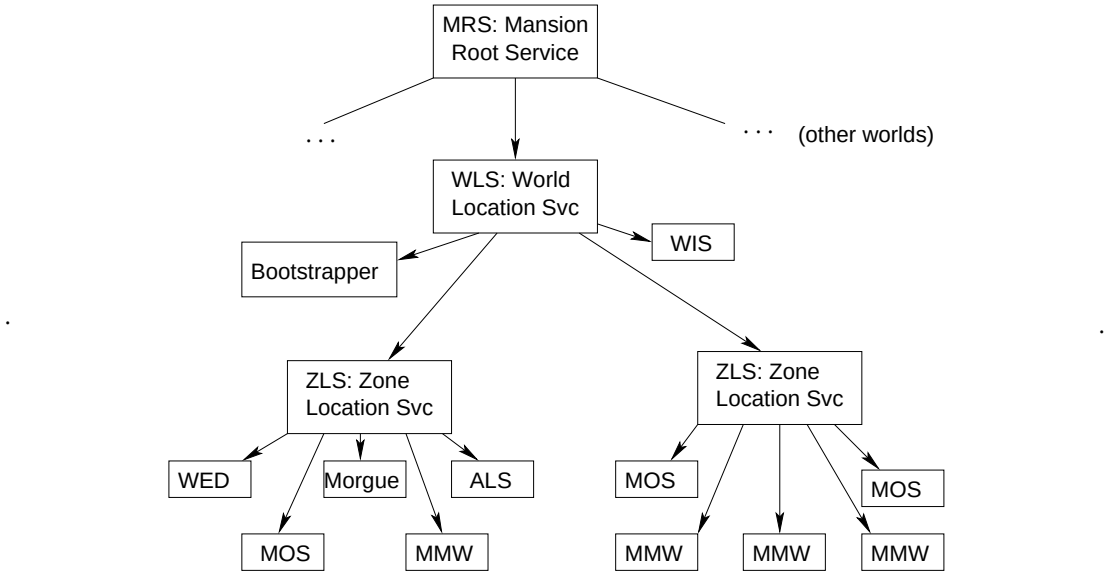

Fig. 5. An overview of the Mansion Location Service

The WLS contains references to world-wide accessible services. Objects and per-zone services are registered in the per-zone zone location service (ZLS). The MNS and the MRS are currently managed by the Mansion developers. The WLS can be found via the mansion root service (MRS) using a using a special handle, *<WorldID>_WLS_0_0*. Some notes:

- Each agent gets a unique identifier (an *AgentID*) at world entrance. This identifier is generated by an ALS. The AgentID contains the ZoneID of its ALS.
- A handle is a location-independent identifier for a service or object. It uses a simple naming convention: it contains the base32-encoded ZoneID, followed by underscore, followed by the name (its IDL *type*) of the service, underscore, an and an *instance identifier* (in case there are multiple objects of the same type in a zone).
  The location service associates a handle which a (set of) contact record(s).
- Knowing an agent's identifier and given the naming convention for handles, it is straightforward to create the handle of a specific known service, such as an ALS. As an example, agent (AgentID) 6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_12 can be resolved in ALS 6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_ALS_0_0.

Using a process of incremental lookups based on straightforward naming conventions, any object or service's handle can be resolved, typically starting with a ZLS. Most handles can be

constructed in a similar way. A **resolver** library linked in with Mansion middleware processes can resolve handles based on their type, and caches results. Details on naming conventions used to resolve handles are provided in appendix 5.

3.6. Putting it all together: overview of a world and services

Fig. 6 shows an overview of a world, including the services and the zones described in this chapter. Shown are a world zone, containing the world location service (WLS) and the basement. Further, a world entrance zone and a regular zone are shown. The components of the location service (WLS and ZLS) are also shown, including the Mansion name service (MNS) and the Mansion root service.

The location service of Mansion is distributed over multiple zones. The contact addresses of rooms (RMOs) and objects, and MMW processes hosting agents, are registered in the per-zone zone location service. Location services may be single-instance per zone, or may be replicated. Replication may be useful if a zone is large, spread over a large geographic area, or simply as a primary backup mechanism. Current services are non-replicated, but earlier work on replicated objects can be applied [107,26].

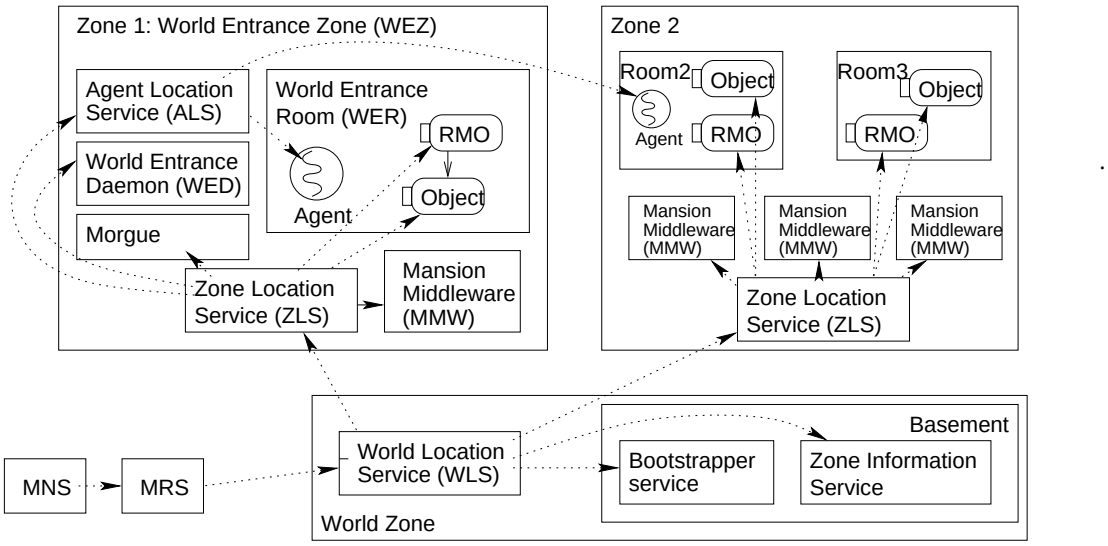

Fig. 6. An overview of important services in a world. Dotted arrows show ‘‘pointers’’ registered in a location service. Zones (including the world zone) are clearly shown. MNS and MRS are global services used to find the WLS.

The location service is hierarchical. The WLS can be found using the MRS, and the ZLS can be found using the WLS. Object servers and middleware processes can be resolved on the ZLS using a handle that can be constructed using a straightforward naming convention based on the zone’s ZoneID. Similarly, an agent’s (home based) agent location service can be

found using the ZoneID in the agent's AgentID.

A world can have multiple world entrance zones; zone 1 is one of them. The agent in zone 2 is managed by the ALS in zone 1; had there been another agent, it might have been managed by an ALS in a different WEZ (not shown).

3.7. Discussion

The sections up to now discussed how logical/visible elements of the paradigm are mapped onto zones, and how internal components are used and distributed. Combined, it shows how worlds are distributed.

Zones create a clear mapping between a logical model and an underlying physical model. All middleware processes in Mansion that belong to a room must be a member of one zone, the room's zone. A room and its content (objects, and agents accessing objects in a room) may only be physically distributed in the room's zone. As a result, it is clear which physical substrate—processes on hosts, managed by a single administrator—a room is distributed or replicated on.

By ensuring that content in a room is only distributed within a room's zone, a clear model of how logical concepts are mapped on physical concepts is achieved.

Entities can be located in a distributed location service. Each entity has a handle that contains a self-certifying ZoneID. Using this handle, contact addresses for the entity or service can be resolved, and the entity can be authenticated. The mechanism allows for decentralized, scalable management of zone membership (by a zone administrator), and avoids that components need to trust central authorities or components in order to authenticate processes as members of zones.

The following sections describe how, in a given world, properties can be associated with zones. Based on this, agents and other parties can decide how much they trust a given zone or entity. The chapter ends with a discussion on controllability and scalability of worlds from an administrative perspective.

3.8. World management

Most of the remainder of this chapter will deal with scalability and controllability issues. As outlined in Chapter 1, an important concern with the design of Mansion was to ensure that a system is *controllable* on the one hand—in terms of aspects that a world owner needs to control to maintain a consistent world structure—while also ensuring *scalability*—avoiding that central administrators or components become overloaded when a world grows—and *autonomy*—ensuring that world owners cannot control every little detail in a world, and that zone, content and agent owners retain autonomy to make decisions or to manage content without interference from central parties.

This section will describe the components needed to control aspects of a world, and subsequently discuss scalability and (decentral) autonomy related aspects.

3.8.1. The world design document

The primary goal of Mansion is to structure worlds . The basic structuring concept is that of rooms containing objects and agents, with hyperlinks connecting the rooms. Worlds are closed, so that the hyperlink structure and the content of the world can be adapted to the application. All entities are described using—again, application-dependent—attribute sets, using which agents can find their way.

Maintaining a coherent, logical structure in a world can be difficult. There exist many systems to date with an open structure—that is, few rules exist. This includes the World Wide Web. In such systems, applications, users, agents, and their programmers are essentially relied upon to ‘‘make sense of the mess’’ at the application level. Agents can help do that to an extent, but in very large systems the task is daunting [127,84].

Mansion is different. Mansion allows world designers to create rules to structure a world. When, rooms, objects, and information are added to a world, the world designer requests their maintainers to adhere to these rules. Structuring worlds has benefits for programming applications and agents, and for users. This way, they know what to expect when entering a world.

Mansion allows a world designer to impose an application-specific structure on its world. The **world design document (WDD)** is the central tool to help in this process. Important instruments are the attribute sets that define properties of entities in the world. An important responsibility of the world designer is to create a hyperlink structure. The WDD allows for defining attribute sets, object *types* (interfaces) and hyperlink topologies (topology constraints) that suit a given application best.

For example, consider an auction world. Rooms may have certain topics, like cars, paintings, or energy. Then, rooms (hyperlinks) may have an attribute ‘‘topic.’’ As an example for object attributes, a multimedia world may contain a *MediaStorage* objects for audio, image or movie files; a ‘‘mediatype’’ attribute may indicate the type. Each object may contain the same content in different formats, such as raw or JPEG images, or MPEG movies with different resolutions or different codecs. In a multimedia world, it makes sense to advertise properties like these in attribute sets.

World designers can set rules on these aspects in the world design document. The WDD can be parsed such that the MMW and RMOs in a world can enforce some of its rules automatically. For example, an RMO can check whether a registered AS confirms to the WDD’s rules, and the middleware can check the hyperlink constraint policy at the time of migration or registering a hyperlink[13]. The WDD is intended to be program-readable; it may be that updates to the WDD (e.g., changes to the hyperlink constraints) are distributed at regular

[13] If checks are made at hyperlink registration time, checks need not be made at agent migration time.

intervals. The WDD is signed by the world owner, so its authenticity can be verified by all members of a world.

The WDD currently contains the following:

- **Attribute set templates** for hyperlinks, agents, and objects. *EntityID* and *EntityType* are mandatory attributes in every world. Other optional or mandatory attributes (for example, ''name'') can be defined by the world designer per EntityType. In some cases, attributes may come with a list of possible values. For example, a world designer may define an attribute ObjectType with the object types *FileContainer* or *MultiFileContainer.* Non-limitative or optional attributes may be left empty. Mandatory attributes in which case *must* be filled in when the entity is registered in an RMO.

- **Object types list and IDL definitions.** A set of object interfaces (types) may be defined to aid agent development. Mansion comes with two object types, *FileContainer* and *MultiFileContainer*, as well as a number of Mansion services, currently implemented in C++. A given world may provide additional objects, e.g., through a software repository. The IDL definitions of all objects should be included in the WDD to enable programmers to compile stubs for these objects, possibly in different agent programming languages[14].

- **Hyperlink constraints.** Depending on the world, it may be useful to constrain the way in which hyperlinks are created. Hyperlink constraints generally describe how zones may interconnect, not how rooms may interconnect, since the latter is not enforceable at the world level[15]. Example hyperlink constraints are described in appendix 2. Besides hyperlink constraints that can be directly enforced by middleware processes, a world designer can document general constraints or intents regarding the world's hyperlink topology in external documentation. Examples of additional instruction may be ''prevent cycles,'' or ''zones should have only one entry room, no deep linking.'' Adherence cannot be directly enforced, but a world designer can use agents to verify these constraints. Because a world owner can remove zones from a world, compliance to the rules can be enforced.

- **Security constraints.** A WDD may contain certain security-related parameters. An example are cipher suites that define the cryptographic channels between middleware processes. Other security constraints may be security clearance levels, which may be associated with agents at registration or world entrance time. Such levels may influence what rooms or content an agent can interact with. We have not implemented such levels, but object access control lists exist that allow for dealing with such levels in principle, assuming objects understand the clearance level or role information assigned to agents.

[14] Note that the WDD should typically be accompanied by more detailed programming documentation, such as notes on error handling for the objects; these may be provided externally, e.g., through a website. There may be stub repositories per world, but these are not currently implemented.

[15] In theory, constraints on how zones interlink may in theory be enforced by the Agent Location Services of a world, which can effectively block agent migration by refusing to update an agent's location address to a given zone.

- **Configuration aspects.** A WDD contains a number of global configuration parameters. Examples are the maximum size of an agent container, a list of allowed programming languages or binary agent types (e.g., Linux 64 bits and 32 bits), and a maximum *time to live* (in seconds) that agents are allowed in a world. The latter parameter may also be defined as a runtime parameter set at world entrance time than fixed in a WDD. Finally, a WDD contains a declaration that specifies whether the *jump* primitive is allowed in the world.

In appendix A, an example of a WDD is shown as it is currently used in the (default) Mansion setup. The WDD currently has no complete formal language (hence it is not called a world design *language).* It consists of various sub parts which may be interpreted differently for a different purpose (e.g., the IDL). Mansion comes with a simple WDD parser that splits the document into relevant parts and places them at a well-known location in a central *bootstrapper* service for downloading by middleware processes.

3.8.2. Managing zone properties—the central zone list

All zones in a world are in a world-wide zone information service, in a list called the **zone list**. The zone list is simply a list of ZoneIDs with, for each zone, the description as accepted by the world owner. The description may be vetted by the world owner before registration, depending on the world. Using the zone list, properties of the zone (such as its owner / administrator) can be looked up so agents and other parties can determine whether they trust it. If a zone is not in the zone list, it is not part of the world. The world owner decides what zones are in the list; depending on the world, the world owner may have extensive verification procedures or may simply accept any information as-is.

The role of the world owner is similar to that of a certificate authority (CA) in a Public Key Infrastructure. By signing the zone list, the world administrator *binds* information to (public) keys, in this case ScIDs. By controlling information about all zones, a world administrator acts as the *root of trust* for zone information in a world. Like a CA, a world owner has to be trusted by all users of a world. This trust is implicitly established when using a WorldID to bootstrap a world or injecting an agent in it.

Zones may be associated with various properties using **zone descriptions.** Mansion does not pre-impose any description. Properties need be associated with zones only if this is needed or useful for a world. A typical zone description contains a zone's name and its properties, such as its owner/administrator's name. If properties are defined for a zone, they should be applicable to all zone member processes or the systems they run on.

Zone descriptions are attribute-value pairs describing properties of the zone. Examples may be host configuration details, if relevant. For example: ''OS=linux'' or ''OS-version="2.4.22".'' Another example is where a DNS name is encoded in a zone description, for example, *amazon.com.* Zone descriptions may also contain a reference to a Service Level

Agreement (SLA)—digitally signed by the zone owner—that specifies properties that may impact performance, security, or privacy, such as: ‘‘we jail all processes’’ or ‘‘we guarantee to remove all temporary data stored by an agent after exit.’’ Such a reference to an externally defined SLA should be accompanied by a hash of the SLA, to ensure the zone owner cannot change it without the world owner’s permission.

Zone properties may be optional or mandatory. An example mandatory property may be a common name (CN) or a company’s DNS name in a commercial world. Zones are not geographically constrained, so their properties need not contain location or organisation information, but they may; physical properties (e.g., hardware information) may also be contained. Alternatively, zones may simply act as a grouping of processes that is suitable for authenticating the zone, without any additional information (Sec. 3.2.1).

From a technical perspective, open, low-barrier entry, loosely-defined systems can be constructed just as well as highly secure, constrained and tightly controlled ones, although the latter system’s zone registration processes will be much more complex.

3.8.3. Trust and agent migration

The concept of zones, zone ownership, and zone descriptions exist so that control over and verification of zone properties is possible if this is needed for a world. Due to the zone list, each zone in a world is known, and it is clear what properties it has. Typically, it is thus known what administrative entity (zone owner) is responsible for a zone, so that other parties in a world can decide if they trust it.

Trust is particularly important in mobile agent systems, since agents that run on a machine are vulnerable to attack by that machine or the software or users on it. An agent must migrate to a zone before it can enter a room in that zone. An agent can request information about the zone in which it runs, and it can request information about the zone in which a target room resides. If an agent communicates with another agent, it can request the zone in which the other agent resides. This information can be used by agents to reason over trust related aspects.

Agent owners may not trust all zones equally. Agent owners can create **‘‘trusted zone lists’’** that allow their agents to avoid untrusted zones; their current (trusted) MMW can check this list in their AC, and refuse to migrate and alert the agent if it intends to migrate to an untrusted zone. The agent can override this decision if needed.

Note that an agent owner who trusts a zone, may not necessarily also trust all content in it. But it will typically trust this zone’s execution environment (the host that it runs on, which is part of the zone) not to tamper with it—or decide to take the risk.

3.9. World management

The zone list is a central, trusted *authoritative* list that contains information about all zones in a world. The reasons for having a world-controlled zone list are:

- The world owner needs a way to include or remove specific zones in the world. The zone list contains the authoritative list of zones in a world. Zones (ZoneIDs) not listed in the zone list are simply not a part of the world.
- The world owner needs a way to impose and—if needed—exert control over properties of the zones in a world, such as security or configuration properties.

It is up to a world owner to decide how important (validation of) zone properties is and, if needed, to collect (and verify) zone properties at the time when a new zone is registered. Before a zone is entered in a world's zone list, a world owner can impose requirements on any party that wants to join the world. This may be loose, or it may include strict verification measures on zone owner (contact) information, or contracts or (service level) agreements regarding zone properties. This may ensure that the zone owner can be held liable if zone properties do not correspond to those specified in the zone description.

The basic implementation of zone registration currently consists of a script that emails a ZoneID with the zone owner's email address to the world's owner (the world owner's email address is found in the bootstrapper service). This way, a zone owner can be contacted by a world owner, for example, to obtain more information. In an ''open'' world without meaningful world entrance control, a central script could take care of instant inclusion of the ZoneID in the zone list, possibly with some email and key verification procedure at registration time.

Note that Fig. 4, shows a zone certificate that also contains a **zone (member) predicate**. A zone predicate is an optional way for zone owners to describe (additional) attributes of a zone (member process) in the zone (member) certificate, in addition to the general zone description. These properties are not checked by the world owner. Zone (member) predicates can be used to specify details about zone members which may change over time, or which may be zone-specific or not critical to world membership. For example, there may be OS-specific details which may differ for members of a zone or which may change over time—for example, the OS of a host on which a zone member process runs. The zone predicates provide some extra flexibility, making it possible to manage and evaluate (at connection time) certain details decentrally, in addition to the world-wide zone descriptions.

3.9.1. Adherence to world rules

For adherence to the world design rules, much the same holds as for other types of control: the world owner can remove zones from the world if these do not adhere to the rules.

Adherence to the WDD is important to allow consistent structuring of worlds. At the same time, zone owners should be autonomous in the content they manage—in as far as they adhere to potential rules set by the world owner. Worlds also need to be scalable. Autonomous and relatively independent zones are important so that they can scale up or down as needed, and remove or add zone member processes, content, or hosts as needed.

Maintaining a coherent, logical structure in a world can be difficult when providers of content (rooms, objects) are autonomous entities who host their content. The requirements of scalability and autonomy mean that the world owner does not have absolute control. Such control would not scale from a technical and administrative perspective, and would interfere with autonomy of zone and content owners.

The deployment model for content in zones is not unlike the World Wide Web, where people are free to add content (as they are in Mansion), but the Web does not attempt to enforce *any* structure on the world. In contrast however, Mansion worlds, developed for a certain application or goal, may impose constraints on content or structure. In practice, world owners may require zone owners to agree to specific constraints or rules before they are allowed to join the system. Periodic checks and peer pressure can ensure that content owners stick to the rules, with removal from the world being the ultimate sanction.

3.9.2. Power and consequences

A world owner cannot directly control what middleware processes and objects in zones do or not do. But if zones do not adhere to a world's rules, a world administrator has one ultimate coercive power: she can remove a zone from a world's zone list, effectively removing it from the world.

MMW processes in a world are expected to check (update) the world's zone list regularly, will only accept agents migrating to and from a zone that on the zone list. The agent location service (ALS) also does not allow updates of an agent's contact information to a zone that is not in a world's zone list (Sec. 8.3).

If a zone owner does not comply to the world's rules, this can be detected, or may be reported upon by other world members. It can then be removed from the world.

3.9.3. "Dark worlds"

Zones removed from—or unsatisfied with—a world, are free to start or join another world that allows conflicting content. Worlds are completely independent of each other. Suppose a zone owner provides illegal content, is removed from a world, and starts a new world. Does this pose a problem? The only thing that can be done then, is to remove these worlds from the Mansion root services that people use to bootstrap a world. Mansion can employ censorship in that sense, as far as the *official* system is concerned. (Similarly, IARA can forbid DNS

names like www.childporn.com in the official DNS naming system). However, a ‘‘dark world’’ with alternative MRS and MNS servers or even without such services can easily be constructed.

Creating ‘‘subworlds’’ or even ‘‘subversive worlds’’ is thus clearly a possibility. Since worlds are disjoint, autonomous, and independent, it is straightforward to create ‘‘hidden,’’ ‘‘secret,’’ or ‘‘private’’ worlds. This may be a morally problematic result of a system that is designed to create independent application-specific worlds. It should however be noted that this problem exists for all systems, even including the Web which—with a different underlying DNS system—can look completely different in a hidden or secret instantiation.

3.10. World entrance policies

The world entrance zones (possibly subject to rules imposed by the world owner) can implement world entrance policies, and the mechanisms to enforce these policies. For example, a world entrance zone may be accessible through a Web site that offers pre-compiled agents for certain tasks to registered users. Such a web site may require payment to enter a world. A typical scenario is that the owner of agent may have to preregister before he/or she can inject agents. Scientific users may require a (Web) interface where agents can be uploaded (or possibly reused) with different initialisation files. Conceivable are agent composition or workflow orchestration toolkits for multi-agent tasks.

Every world (or even each world entrance zone) may implement its own **world entrance policy.** For example, an agent owner’s registration may result in an agent being flagged by the world entrance daemon as belonging to a certain category, such as ‘‘ivory,’’ ‘‘gold,’’ or ‘‘platinum’’ agents. Alternatively, the registration procedure may require strict identification of an agent’s owner, or it may allow anonymous or pseudonymous access, where a mapping between an end-user’s identity and a ‘‘pseudonym’’ is generated at injection time.

Internally, every front-end uses the same agent submission procedure and mechanism to submit an injected agent to a world entrance room. A world entrance daemon securely binds agent ‘‘identity’’ (or, if applicable, its role) information to an agent at submission time, by storing it in the agent’s Agent Container (Sec. 8.2.6), along with other information such as the agent’s time to live, the maximum AC size, and the AgentID obtained from registering the agent with the ALS.

Currently, the access control scheme for world entrance consists of simple identification of an agent’s owner by means of her public key. A 32-byte identification string (essentially, the agent owner’s ScID, or *AgentOwnerID*) is placed in a special file in an agent’s agent container, and signed by the world entrance daemon. The AgentOwnerID assigned to an agent is opaque to the MMW and can have several meanings, such as a security level or as a payment scheme identifier. The string is only interpreted at the application level, that is, by objects, and rooms. It is passed to objects when these are invoked by the agent. Based on this ID, an RMO or other object decides on whether access to the room is allowed. Access control and

semantics of an agent identifier are up to the application; the mechanism is flexible enough to meet different application scenario's. Different services can be provided at different service levels or cost if required: online tracking of agents, pseudonymity, premium access, etc[16].

In terms of control: a world owner can accept or remove a world entrance zone like any other zone. The world owner and the world owner should agree on the types of access policies permitted and enforced. Contractual agreements may have to be in place before the world entrance zone is added to the world. Depending on world requirements, the world entrance rooms may have to enforce certain constraints, such as a maximum number of agents that may enter a world at a given time. Also, world entrance rooms may have to link to specific other rooms in the world so that agents can reach the whole world from it. Finally, SLAs may be in place.

3.11. Scalability

Using different independent zones in a world provides scalability and security advantages compared to where a world owner manages a central infrastructure for all services. With increasing scale, keeping up with the capacity of the infrastructure and the administrative load of managing a single infrastructure could soon become infeasible.

Technically, it is also important that load is spread; if interest in one part of a world increases, would this impact central parts of the system (e.g., world location service), world entrance zones, or even unrelated zones disproportionally?

A related issue is economics. How is the cost to pay for scalability, and the resources required for that, spread over the participants of a world? This section will also discuss some aspects related to that.

The basement and the location service. For very large-scale worlds, world management and the central infrastructure in the world zone can become a bottleneck. From an administrative perspective, a world owner should not have to worry about details of how zones are managed internally.

From a technical perspective, the capacity and availability of the basement and the services therein, including the world location service, is important. The location service of the world is hierarchical. The zone location services that manage the contact records of entities in different zones are distinct, and managed by the zones, separate from the world location service (Fig. 6). Thus, as particular zones may grow in size or popularity while overall popularity of the world remains constant, the load on the basement should remain stable, while only the load on the more popular zone location service(s) increases.

If a world's overall size increases, so will the load on the basement, since more agents and more MMW processes will request information from it to locate zone location services or

[16] When nonstandard attributes (e.g., *roles* or *payment* types) are stored in the agent's AC, the world's rooms and objects must obviously be aware of these; the world owner should specify design notes or guidelines on this.

ALS addresses. (Note: the resolver library used in Mansion uses caching to avoid having to resolve an entity's address repeatedly).

In a hierarchical location service, the (potentially varying) load is spread over the different location services of which it is composed. Concretely, larger and more popular zones will attract more traffic to their location service. First, because more agents will migrate to it, which leads to more requests to find a MMW process to migrate to. Second, lookups to resolve resources within that zone involve the zone location service only. Objects will generally be resolved from within a zone, as only agents in the same zone can bind to them. Thus, requests within a zone will generally not impact other location service nodes.

Zones. Internally within a zone, many services need to scale up as its popularity rises. Capacity for hosting agents can increase by adding MMW processes on new hosts to the zone. Some objects or services (with often accessed read-mostly data) may need to be replicated. This also implies adding more processes (e.g., to host replicas of the service) to the zone.

Adding processes to a zone is done decentrally by the zone owner, and does not require world owner intervention. Most services in a zone, such as object servers managing objects and MMW processes managing agents, are completely internal to the zone and can be created (as new zone member processes) without having an impact on other parts of the world.

As for replication: experience has shown that highly available, large-scale replicated services can be built [26,112,2]. The lessons learned can be applied to RMOs, objects and services (implemented as objects) in Mansion. For location services, related issues have been explored for the Globe location service infrastructure [17], DNS, and for Mobile IP and mobile telecom services. Although replication for location service nodes and objects is not currently implemented, chapter 7 will describe some options for replicating objects.

World entrance. Another potential bottleneck for scalability is world entrance. All agents must enter a world through the world entrance system, so scalability of the system as a whole is impacted by the number of agents that can enter this way.

As the number of agents that enter the world increases due to a *decentral* process of people adding content to the world and people getting interested in using the world, the load on the world entrance daemons, agent location services, and Morgue services in a world increases. It is intuitively clear that one single world entrance room—potentially containing services that help agents on their way, e.g., a yellow pages service—does not scale to huge numbers of agents. Such a room (its RMO and other objects) would have to be heavily replicated[17] over a large number of machines in the world entrance zone to handle all incoming agents.

[17] The room's RMO's state may be difficult to replicate (its state needs to be updated often as attribute sets are written to it for all entering agents and removed for all that leave). It could be considered to change the semantics of world entrance rooms such that in a world entrance room, agent attribute sets are not shown in the RMO; however, creating such an exception would break the conceptual model. Besides, agent attribute sets have a function, such as that other agents can see who enters the world and, possibly, make contact.

Scaling up a world with a single world entrance room would clearly be a challenge. A world with multiple world entrance rooms in one world entrance zone is in better shape, but in this case problems may arise with the number of agents that have to be managed by the world entrance daemon, Morgue and agent location service of that zone.

Mansion solves the scalability issue of world entrance rooms by having multiple decentrally managed world entrance zones each containing one or more (equivalent) world entrance rooms. World entrance zones are technically and organisationally separate from other zones.

The different world entrance systems (a **world entrance system** consists of a world entrance zone, one or more world entrance rooms, the agent injection system—WED, morgue—and additional systems and procedures such as for registering agent owners) are managed decentrally. (How agent owners find the world entrance zone is outside the scope of Mansion, but probably the world entrance system manager will announce the means to register with and inject agents in the world to the outside world by means of, for example, a Web site. Having a choice of world entrance zones can help an agent owner to select the best one for its purpose. World entrance system managers may compete with each other to present attractive front-ends and services in their world entrance rooms to users; details are outside the scope of this thesis).

The world owner may have Service Level Agreements (SLAs) with world entrance zones. As the size or popularity of a world increases, the capacity of the more popular world entrance zones may have to be increased to meet the SLA. Presumably, world entrance zones charge for their services in some way, to pay for the cost incurred. Further, as world's popularity rises or if existing world entrance zones struggle, new world entrance zones may be added such that more agents can be served in total.

3.11.1. Economical aspects of scalability

Keeping a set of reliable services for agents to enter (and exit) a world and keeping track of agents using a reliable world entrance system is not cheap, and is probably infeasible without some sort of payment or revenue model associated with it. Similarly, as a world grows, demands on availability of the basement will grow. This section explores and discusses some aspects related to whether revenue and business models can support scalable world deployment.

As the size or popularity of a world increases, so does the load on the central (basement) services, on the world entrance zones, and on the individual zones in a world. It is important that if an increase in load requires investments for scaling up, that these investments are ''fair,'' that is, a service or zone that does not contribute to the increased load, should not have to scale up disproportionally. If required investments to handle increased load on services are skewed compared to where the load is generated, then even if technical scalability is possible (which we assume), the required investments may not be made and the system may end up with bottlenecks in practice. Therefore, it is important to see whether an increased load on

some part of the system is proportionate to the potential revenue that can be made by this part of the system, and if economic models can be conceived which can sustain a (large scale) world.

The potential number of individual objects and services in a world may be large compared to the number of zones in a world. Objects, services, and agents may also be subject to a high degree of ‘‘churn,’’ that is, subject to the dynamics of disappearing and (re-)appearing nodes and processes in a system. This puts a large load on the location service system. For objects, this mainly concerns the zone location services. For agents, most of the load is on the agent location service; the load on an ALS is related to the number of agents that entered through the ALS's world entrance zone and which are in a world at a particular time. Churn of services may impact both global and zone-relative services. At least some kind of replication of the location services (ZLS or WLS) may be required—not just to cope with load, but also to ensure availability and reliability of the system. This incurs cost for machines and administration.

Depending on how many agents visit a zone, computational resources are required to host the agents. Whether or not providing computational power to host agents is economically feasible depends on the application. A few examples:

- Users who submit agents may be willing to pay for the service provided or the data collected; this can be the basis for business models that support the resources needed for the framework.
- Universities or medical centers may have sufficient incentive to fund the resources required based on the prospect of setting up (medical) trials, research, or collaborations in ways which were infeasible using the Web—the added value of the applications may be worth the cost.
- At the other end of the spectrum, volunteer resources (end-user machines) may be added to a zone to host content and agents, like Web pages may be added to the Web. NGO funding and/or small world entrance fees to cover the cost for managing world entrance zones may sustain the costs.

A scalable system should provide fairness: a zone consisting of a huge number of objects, middleware (MMW) processes, and agents, should not place an undue or disproportionate load on an unrelated part of the system. The hierarchical approach of spreading location service(s) over the system, combined with the fact that rooms are accessible from their own zone only, ensures that (location) services and objects of the popular zone will be loaded specifically when popularity of a zone rises. If a given zone is popular, the increased cost of maintaining it is the price to pay. The hierarchical approach of spreading load thus seems to provide the required fairness, as other zones are not impacted. Interestingly, had we chosen to use a less strict model on agent migration where agents could have accessed rooms from different zones, fairness may have been less predictable.

The world entrance zone has another position. The load on a WEZ may increase as popularity of a world as a whole increases. Although the WEZ may have to invest in dealing with the resulting increased load, revenue (e.g., due to increased sales) may go to regular zones in the world. One can consider that agent owners will pay the world entrance zone for entering the world. This payment may cover the cost of some or all of the resources used in the world entrance zone. (The WEZ may use a registration fee, a fee depending on how long agents use the system or on how often they migrate, or on the (maximum) size of the AC, which is stored in the morgue.

As an alternative, the WEZ may bill zones in the world, e.g., corresponding to the number of agents that visits them; aggregated migration statistics may be obtained from the ALS to indicate which zones in a world are popular and must be billed. It should be noted that this may cause privacy concerns.

A very different model is where the world entrance fee has to cover all costs in a world, including the cost of deploying regular zones, who thus bill the world entrance zone—this may occur in cases where content is provided by non-profit or academic institutions. In addition, the world administrator may demand a share of WEZ revenue, irrespective of who payed for this revenue.

Presumably, all cost will eventually be payed by agent entrance fees or or agent owner registration costs, or as a share of revenue by zones who provide content. Although a detailed discussion is outside the scope of this dissertation, it seems that mechanisms (in particular, accounting for use using the ALS) are available to WEZ administrators to account for usage of the world. This can help construct feasible economic models to support scalable world deployment. Accounting for use, billing for placement of a hyperlink in a (world entrance) room, payment for registration of a zone in a world, etc. are conceivable instruments. It appears that economic models can be mapped on the deployment model of Mansion.

3.12. Examples

To close this chapter we describe a few example worlds to illustrate the aspects discussed in this chapter.

A single-zone world. The content of a single-zone world will be provided by the world administrator, or at most by a few content providers who then provide the content through the world administrator. There is little complexity here, as scale is limited. The world owner has complete control over all rooms, objects, hyperlinks and content in the world.

A multi-zone commercial world. Closest to other examples in this thesis, an e-commerce world allows registration of zones for shops that sell music, movies or images. Registration comes for a fee, and require that prospective zones show their registration before the shop's information is registered in a zone description. The world services are payed by these fees. There are three world entrance zones, with world entrance rooms specialising in

movies, images and music, respectively—with cross-links to the world entrance rooms of the other zones. The world entrance zones get payed by zones, with accounting done by checking the time that agents spend in each zone using the agent location service. Agent owners can register once and presumably pay a small fee once; the assumption will be that the investments needed to host agents in a world will be payed by music, image or movie sales. Some shops may provide confined rooms to search expensively priced rare music or movies while other shops may allow agents to stream music home (pay per minute). The world owner, together with world entrance zones, spends a lot of time optimising the hyperlink layout to ensure the world allows for finding the different shops easily without bias for the larger shops. Agents or services provided by world entrance zones may help locate shops with specific content.

An open, huge-scale many-zone world. Consider a very large-scale, World-Wide-Web like world, where anyone is allowed to create zones and content and place it in the world using a simple, automated and fairly insecure procedure. In this example, all content (room) providers run their own zone (containing the middleware services required for hosting agents and rooms) similar to how in the Web, many organisations run their own Web servers without any central control. The world owner does little, for administrative scalability reasons, but also due to reasons of ownership and trust: not everyone will trust the world administrator, even though they use the world, and also the world administrator is not interested to host thousands or even millions of rooms for various parties; this would simply not scale. Therefore, the world administrator primarily ensures that the world location service (and the basement) stay up and running and that zones can be added and removed (automatically) from it.

A secure banking world. Imagine a banking world, where zones may only be added using a (costly and presumably difficult) highly secure procedure. This procedure is needed to ensure that only legitimate banks enter a world. Also, the zone keys (ScIDs) of these bank zones must be properly authenticated. SLAs ensure that liability issues are covered and establish a baseline (mutual) trust between all contributing parties. Certification of zones (banks), their systems, and their personnel may be required. Only authenticated and authorised agents will be able to enter the world; agent (owner) registration and world entrance will be tightly controlled. The underlying network connections need to be encrypted with the strongest cryptographic standards, if not also run over leased lines.

Clearly, the above worlds differ in a number of (orthogonal) aspects. The examples illustrate why Mansion must permit different tradeoffs to be made, translating into different levels of control depending on the scale and purpose of a world, as was discussed in this chapter.

3.13. Summary and discussion

Chapter 2 of this dissertation explains Mansion's logical model. Chapter 3 explains the way in which the logical model is mapped on the physical model (by means of the zone concept), and the world description document. Combined, these two chapters describe the basic model

of the Mansion paradigm. We summarize and discuss it here.

Rooms and content are deployed within a zone; an agent that wishes to access a room or its content has to migrate to that room's zone, since objects are only accessible within a zone. Agents can interact with each other anywhere in a world. Agents can communicate with other agents using secure, migration-transparent connections. However, in a confined room, an agent cannot communicate with the outside world; its communications with the outside world are cut off, and the only way in which to export information from a confined room, is by passing it through the room's guardian agent.

Rooms and objects in those rooms are only accessible by agents in this room. The object servers in which objects, including the RMO, run, are only accessible to processes in the room's zone. Mansion's migration model is simple and clear. Upon following a hyperlink, agents are always restarted. Mansion only supports *weak migration*, meaning that agents are restarted retain no execution state or memory upon migration. Except for what agents store in their private agent container, nothing is kept when an agent migrates. In a confined room, an agent is not allowed to write to its AC.

Hyperlinks are logical links to rooms; agents are normally not aware of the physical location of entities. World owners may impose constraints that restrict migration (through hyperlinks) between particular zones. For scalability, objects may be replicated, and zones may exist on multiple (middleware) processes on several machines in a zone. These aspects are invisible to agents.

A world can consist of a large number of zones, each containing a large number of rooms and objects, or of a single zone. A world design document describes properties of the world, such as object interfaces, configuration aspects, supported agent programming languages and inter-zone migration (hyperlink) constraints. Attribute sets describe components (agents, objects, hyperlinks/rooms), and allow agents to find their way.

The world owner controls what zones are part of a world, and may impose constraints on (world entry) zones in the world and, to a limited extent and indirectly, over the content of a world, since ultimately a world owner has the power to remove zones from a world. The WDD contains template descriptions of the attribute sets of all entities in a world and hyperlink constraints. Zone owners may lead an agent through a path of hyperlinks, for example to enter (sub)sections of rooms on a particular topic.

Multiple independent Mansion worlds may exist independent of each other. The WorldID is the self-certifying name of a world; it allows users to retrieve and verify the world owner's public key, which is the root of the world and its (implicit) PKI. Using the Mansion root services, a world's WorldID may be used to bootstrap a world.

A zone owner is responsible for managing both the content as the physical infrastructure of its zone; as zones become more popular, content (objects) can be replicated, and MMW processes may have to be added (on additional machines) to handle increasing numbers of mobile agents. Zone members can be added by signing a zone member certificate for the new zone member process using the private zone key.

A zone is created straightforwardly by creating a zone key pair and a self-signed zone certificate, and can be tested ‘‘live’’ outside a world. The only required step to link a zone to a world is to get the new zone’s ZoneID registered in the world’s zone list and to get hyperlinks to its zone entrance room registered from an existing (regular or world entrance) room. How this works depends on the world – the process may be similar to getting a new domain name registered in DNS, or it may be closer to getting a certificate signed by a certificate authority, which may include face-to-face verification—depending on the constraints imposed by a a world owner. In worlds where the world owner places no constraints on zones that may enter, an automatic registration procedure could suffice.

An important property of Mansion is that the world administrator controls a world by controlling the world’s zone information service and zone list. This control is needed to impose application-specific a structure or other application-specific properties on a world. By controlling the world’s zone list, a world owner exert power even over internal aspects of content in a room if needed, since any party (zone) that violates rules on content or structure of a world can be removed from the world. How fine-grained the degree of control is depends on the applications; probably, very fine grained control is normally not needed, works counterproductive as it invades autonomy and, potentially, privacy, and cannot work at large scale, but some degree of control is important.

Mansion starts from the notion that a world owner can design worlds with a specific structure, or specific content, or other properties which suit a particular application. This can only work if some form of enforcement is possible. Enforcement at a low level (e.g., through rules embedded in all the processes in a world) is difficult if not impossible to implement, as individual middleware processes in a world run outside control of the world owner and may simply circumvent the rules, for example by modifying the MMW software. Controlling zone membership of a world is thus a useful coercive means to impose control if needed.

In Mansion, each party who wants to run its own resources can create its own world, and in most worlds may be allowed to create and register its own zone. Within this zone, a zone owner is free to deploy as many rooms and objects as it wants, possibly subject to world requirements. Typically, these requirements only involve high-level, inter-zone hyperlink constraints and zone properties[18].

Depending on how the world is designed, a world may consist of one, a few, or a large number of zones. It is important that a world can adapt to growth and scale over time. A small world may be owned and deployed fully by the world owner, as a single administrator. It can start with a simple world entrance zone managed by the world owner, and grow over time by adding zones and rooms. As a world grows, (professionally managed) world entrance zones may be added to the world; over time, the original zone may be removed, or kept for testing purposes only. As the world grows (or shrinks), (world entrance) zones and zone members can be added or removed, often transparently.

[18] A zone owner may have to indicate certain properties of its zone, for example, properties of the underlying machines used to host agents, or the geographic regions or countries that the zone is hosted on, in the zone descriptions. This may be relevant in some worlds, as such aspects can have legal implications, for example due to data protection regulations.

All processes within a zone trust each other equally. Agents, and other zones, must determine for themselves whether they trust a given zone. Agents can have a ‘‘trusted zone list’’ in their AC to help prevent migrating to untrusted zones. Depending on hyperlink constraints, zones may refuse agents coming from a given zone, or refuse agents from specific untrusted owners. In that sense, users (agents) and content owners of a world are autonomous. Only the world zones and the world entrance zone chosen by an agent owner need generally be trusted fully by the world owner, room/zone owners, and agent owners alike.

This section discussed a number of aspects related to structuring and deploying a world that illustrate how controllability (security) and autonomy requirements are balanced at various levels of the system. Overall, Mansion is a flexible system that balances these requirements well, and allows for various degrees or tradeoffs when balancing control of administrators versus autonomy of users or content providers, depending on the application. Since Mansion intends to support different applications using different worlds, this is not surprising and it illustrates the strengths of the framework.

The discussion on scalability and economics exemplified how business models are likely to support the investments needed when worlds scale up, with fair division of work and cost being possible due to most cost being incurred in the zones where most agents migrate to.

Following chapters

With this chapter we conclude the first three chapters that describe the architectural model and distribution aspects of Mansion worlds. We now turn to the design of middleware components.

Mansion is a layered system. The next chapter describes the lowest layer in the middleware stack, Agent Operating System (AOS), followed by a chapter describing the Mansion communication stack and RPC layer. Following this, chapters describe other middleware components: the jailer and the Mansion object server (MOS) that makes use of the jailer and of the RPC layer.

Next, a chapter describes he main middleware (MMW) process that manages agents (using the jailer and AOS), and the Mansion API; here, all layers and components come together.

Final chapters describe applications of the system and conclude with a discussion.

.

Chapter 4
Agent Operating System

AOS is a stand-alone middleware ‘‘kernel’’ designed to facilitate the development of mobile agent systems. It is described here as it is the basis of the current middleware implementation; following chapters will describe the layers that are constructed on top of it. AOS is used in Mansion as well as in the AgentScape [127] mobile agent platform.

This chapter describes the design and implementation of the Agent Operating System (AOS). AOS provides generic data storage for agents (agent containers), secure agent container transport, and secure communication mechanisms for middleware systems. The main idea behind designing AOS is that if common functionality is provided by AOS, this will help save development efforts for different mobile agent systems. It also should ensure that basic security mechanisms are in place regarding communication and agent transport. Two interoperable AOS kernels have been implemented, in C/C++ and in Java. This chapter presents comparative performance measures of these two implementations[19].

4.1. Introduction

Designing a secure and reliable mobile agent system is a challenge. Over the last decade, various mobile agent middleware systems have been developed to support (mobile) multiagent applications [19,20,51,59,110,115]. These applications depend on a middleware system (called a mobile agent system or mobile agent platform) for functionality such as agent life cycle management, communication, migration, and security. Over time, various mechanisms for security, such as audit trail based security [115], have been designed, but few of these have been adopted in other mobile agent systems. Most agent systems have been implemented in Java. This provides portability and security advantages. [45]. However, for many tasks, support for agents in different languages is useful, for example to allow legacy

[19] This chapter is based on an article that appeared in 2007 [77].

programs to be reused in mobile agents. Few systems support agents written in other programming languages than Java. High-level protocols, for example for interagent communication, have been defined as international standards [21, 70], but not the low-level protocols needed to, for example, ship an agent. As a result, most agent systems implement their own form of agent transport, communication, or security, typically in a (language) specific way.

This chapter presents a different approach. Based on our own experience in constructing mobile agent systems and related work [127,79], this chapter identifies a minimal set of common primitives required for mobile agent middleware systems. Rather than focusing on specific solutions, a portable and generic ''kernel'' is constructed which provides the common primitives required for constructing mobile agent systems. AOS provides support for secure communication, secure agent storage, and agent migration.

AOS is implemented as a separate process which can be invoked by middleware processes, instead of as a library. Having AOS run in a separate address space allows for effective protection against faulty or compromised middleware components. This design has the additional advantage that middleware processes can be implemented in different languages, accessing AOS through a language-neutral RPC interface. Finally, it can help overcome a practical problem for systems that run multiple programs that must be available from the internet: port availability. Using AOS, only one TCP/IP port must be accessible, over which connections to multiple middleware processes can be multiplexed. A simple but effective security mechanism is designed to logically separate AOS resources owned by different middleware processes.

Designed as a portable and language-neutral layer between local operating systems and higher level agent platform middleware, AOS provides interoperability between agent platforms and between different implementations of AOS itself. Different middleware processes can use a single AOS kernel (the process that implements AOS functionality) at the same time. Two AOS kernels have been implemented, in C++ and Java, based on a single specification of the AOS API and the low-level protocols [81]. These two implementations were intensively tested for interoperability. AOS is used in a middleware system implemented in C (Mansion) and in a middleware system implemented in Java (AgentScape).

This chapter is organised as follows. The requirements and considerations that drove the design of the AOS kernel are described in section 4.2. Section 4.3 presents the architectural design of AOS. 4.7 compares the performance of the AOS kernel written in C++ with the kernel written in Java. Related work is discussed in section 4.8, and the chapter concludes with a summary in section 4.9.

4.2. Design requirements

Most mobile agent middleware systems are designed to support specific agent models and programming environments. These systems share functionality. AOS has been designed to provide a minimal common base which provides this shared functionality to mobile agent

middleware systems. By design, AOS supports interoperability, facilitating open and extensible design of agent middleware, and enabling interaction and/or integration with (existing) middleware services.

The commonalities found between agent middleware systems can be broadly classified as: (i) mobile agent (code and data) storage and transport, (ii) primitives for agent life cycle management, and (iii) secure communication between middleware processes (irrespective of what is actually stored in an agent or being communicated). In addition, all mobile (multi)agent systems require security mechanisms that allow for, for example, authentication and authorisation of remote processes, and for integrity verification of migrated agents and content.

The AOS kernel is designed as a stand-alone component with a well-defined interface, that provides flexible and secure primitives for building mobile agent systems. The requirements defined for AOS are the following:

- AOS should provide a minimal set of primitives required for building (secure) mobile agent middleware. AOS should be a robust, minimal layer supporting a broad range of agent middleware systems.
- AOS should be portable and interoperable when implemented in different languages, and reliably run on different platforms.
- AOS should be reasonably efficient, within the expected performance boundaries of (secure) agent middleware. It should not add significant overhead compared to a distributed mobile agent system written with similar (security) requirements in mind.
- AOS should implement mechanism, not policy. AOS should not impose design limitations or a specific model on the mobile agent middleware designer For example, it should not require the designer to adopt a specific deployment or security model (e.g., using a public key infrastructure).
- AOS should be usable without administrative privileges on a hosting machine.
- AOS should be usable by different mobile agent middleware systems concurrently, as well as privately (non-shared) by a middleware system or middleware process.

AOS provides a minimal set of primitives required for building (secure) mobile agent middleware. The AOS design and specification is language and operating system neutral, so that it can be implemented in any programming language and ported to any operating system. It does not impose specific design limitations or a specific model on the mobile agent middleware that uses it. Minimality implies that some mechanisms have to be implemented by the agent middleware itself. This is inherent to the idea that many mechanisms are middleware specific. In short, AOS should be ‘‘lean and mean,’’ and provide only the basics needed for implementing (secure) mobile agent middleware.

For example, host protection and agent life cycle management are handled differently between mobile agent middleware systems. Some systems view agents as processes, while other systems view them as threads running in an agent server. As a result, it is hard to attain a single, simple model for secure agent execution and life cycle management. Also, agent execution may be somewhat operating system specific. Agent life cycle management mechanisms are thus not provided by AOS.

AOS should not rely on external services, as such reliance could effect reliability and performance. Interactions with a remote process can block or fail in several ways. AOS interacts with other AOS processes only for primary tasks, for example to set up a communication channel to another AOS kernel to ship an agent. Management tasks spanning more than one machine or that involve external services are the responsibility of the middleware. This coincides with the requirement that no (deployment) model should be imposed on the middleware: location services, public key infrastructures, etc. are high-level issues and the responsibility of the middleware designer.

Sharing an AOS kernel between different middleware systems, possibly run by different users, on a single machine, allows for running multiple mobile agent systems behind a firewall, with one or only a few TCP ports open on this firewall. In this scenario, it is evident that AOS must isolate resources owned by different middleware processes. An efficient and flexible mechanism is devised to control access to AOS resources owned by different processes (Sec. 4.4.3); isolation is the default but sharing resources is possible.

Language independence is an important reason for designing AOS as a process that runs separately from other middleware processes. AOS provides one or more RPC interfaces that expose its methods to the middleware. A design decision is that AOS should be able to provide several RPC interfaces (each providing a different language binding) at the same time, such that a middleware designer (or different ones) can choose different languages for implementing different middleware components (Sec. 4.3). AOS provides mechanisms, not policy. It hides specifics of its internal implementation from users. For migration and secure channel setup, primitives are provided that allow middleware processes to securely authenticate a remote (AOS) process using self-certifying identifiers. This avoids the need for AOS to know about or adopt a specific public key infrastructure or trust model (the self-certifying identifiers are or provided to AOS, or verified by, the middleware layer that makes use of AOS).

4.3. Architecture

The intended use of AOS is to provide a common base to a range of specific mobile agent middleware systems. This common base should be seen as a kernel component in a layered middleware system design. Higher-level agent middleware can use AOS for agent code and state management, agent migration, and communication. The middleware can extend the AOS layer with middleware-specific components and services, for example, for agent life cycle management, middleware management, and agent naming services.

The architectural model is shown in Fig. 7.

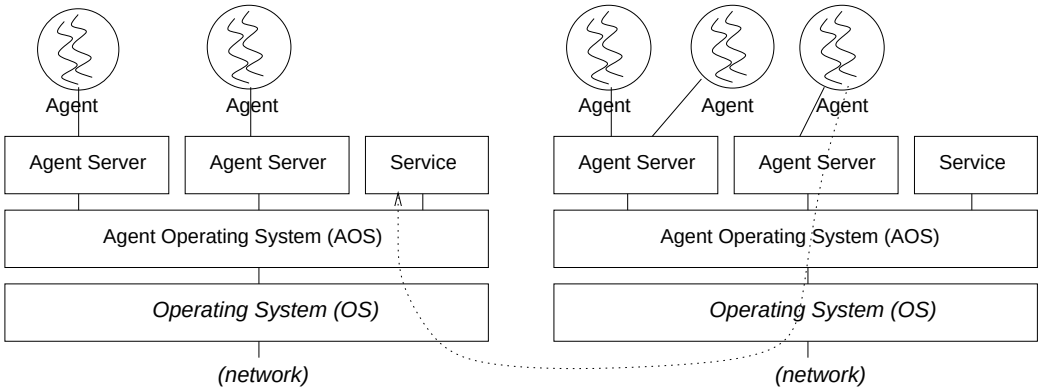

Fig. 7. Example of a layered agent middleware architecture using AOS. This example system consists of two agent server processes and one service (e.g., for receiving incoming agents, or a white pages or agent naming service) running on top of AOS on each machine; this example has some similarities to the AgentScape design. Mobile agent middleware processes communicate with other local or remote middleware components using AOS. Agents communicate with their runtime environment (e.g., agent server), and do not normally access AOS directly. Example flow of an interaction of an agent with a remote service through the middleware stack is shown (dotted arrow).

AOS provides a means to securely communicate with middleware components and services, and to migrate agents to other locations in a secure way. Agent middleware components are distinct processes (see Fig. 7). Agent middleware processes run in a different address space than AOS, AOS methods are accessible through RPC calls. AOS is not directly accessible to agents in general. Agents access a middleware layer that makes use of AOS; this middleware layer presents a runtime environment to agents.

AOS is used both in Mansion and the AgentScape mobile agent platform (Sec. 1.4.6). In AgentScape, agent servers (currently supporting Java and Python agents) are used to run agents as threads. The agent servers provide an API to the agents. In Mansion, all agents are processes that have only a marshalling stub in their address space that invokes methods on an API presented by the Mansion middleware.

AOS has a clear specification for interoperability and portability [81]. The specification describes the API available to higher-level middleware processes, including arguments and semantics

4.4. AOS concepts and primitives

The AOS API provides primitives for agent transport and communication. In addition, AOS provides primitives that allow for protecting or sharing resources owned by different

middleware components. The agent transport mechanism provides integrity protection of agent code and data, and both the agent migration and the communication related methods provide a simple yet highly effective authentication mechanism. The main concepts and mechanisms are described in this section. The API is described later in this chapter.

4.4.1. Agent containers

Agent code, data, and metadata (e.g., owner information, time of creation, permissions, etc) are stored in AOS in a data structure called the agent container (AC). The AC is a set of segments that can contain code or data. Segments are typed to contain code, data, or system data; subtypes can be defined by the middleware system to indicate a specific type of code (e.g., x86 binary code) or system data. Every segment has a name. Segments can be transient or persistent. Persistent segments may not be changed after creation, transient segments may be removed or modified. Primitives for creation of AC and segments, and reading/writing segments are part of the API.

A **table of content (ToC)** contains metadata about every segment, including type, subtype, name, and a *persistent* bit. The ToC is exposed to middleware processes, which can use it to find segments (e.g., by name) in the AC. Every segment has a distinct entry in the ToC, indexed by *SegmentID*. The ToC contains creation and modification dates for every segment.

A *finalize* call is used to synchronise any new content of the AC to disk. This provides for some resilience against AOS crashes (or a *kill* by the owner of the agent system, for example, in a desktop setting where the desktop user reclaims the desktop's resources); the API defines error codes that allow for detection of an AOS restart, and contains a call for re-initialising AOS resources after such an event was detected by the middleware.

The *finalize* call must always be called before migration; it generates checksums over all segments in the ToC, and places a signature over the ToC, to allow for integrity verification of the AC when it is sent to another AOS kernel. The ToC data structure is also used to implement an efficient audit trail mechanism described in Sec. 8.3.2.

4.4.2. Communication endpoints and authentication

Communication-related calls include creation and deletion of communication endpoints (similar to Unix sockets). *Connect, accept, send, receive,* and *select* calls exist which allow for setting up and using secure, reliable, ordered communication channels to these endpoints. These calls are introduced in Sec. 4.6. The *connect* and *accept* calls make use of self-certifying AOS contact records (Fig. 10), that contain the ScID of the AOS kernel.

Middleware processes implement mechanisms to securely exchange information about their own kernel's ScID to their peer process to ensure that they communicate with the correct kernel, that is, the one in their peer's communication stack. In Mansion, such a mechanism is

implemented in the zone authenticated communication (ZAC) layer of the Mansion middleware, which will be explained in Sec. 5.1.5. A given middleware system may also use AOS endpoints directly as endpoints, without end-to-end authentication of the middleware processes. In this case, an AOS ScID or endpoint management system (comparable to a key management system) may be required so that other middleware processes know the ScID of the AOS kernel with which they communicate.

Internal to AOS, a standard authentication protocol based on TLS/SSL is used for authentication and key-exchange, to set up an efficient, secure, encrypted channel to the peer AOS. This protocol is identical to the ScID-based endpoint authentication protocol explained in Sec. 5.1.1. The middleware can specify a cryptographic *cipher suite* for the channel at connection setup time, to influence the strength of the security protocols used by the underlying connection. Other than that, the middleware is unaware of the mechanisms used in AOS for secure channel setup. Connections between the same pair of AOS kernels, with the same security properties (i.e., cipher suites), are multiplexed over a single AOS ''base channel'' which has these security properties. Reusing base channels to multiplex communication channels allows for amortising expensive initial secure (SSL) connection setup times. Agent transport makes use of the same base channels (with similar endpoint records), allowing for safe, confidentiality-protected transport of ACs.

A key property of the AOS authentication model is that the middleware does not have to support a specific PKI when using AOS: middleware processes see AOS contact records, and AOS implements the required mechanism for authentication and secure channel setup based on the information available in the contact record. As long as a correct contact record is obtained (e.g., through a trusted channel), this works securely.

4.4.3. Isolation and resource sharing using roles

Secure sharing of a single AOS instance and its resources by different unrelated agent middleware systems on the same host is enabled by the concept of a **role**. A role is a set of resources associated with a cryptographically protected authentication token (essentially a random bit string) called a **cookie**. AOS creates the cookie securely, and also creates an internal data structure that describes the resources associated with this cookie. A middleware process must pass a cookie with every AOS method that it invokes. Associated with every cookie is a bitmap, called a **role bitmap**, that specifies the methods that may be invoked by the holder of this cookie. The role bitmap is closely related to the resources that can be managed by a middleware process. For example, an agent server which is only allowed to manage agent containers, will only be allowed to invoke AC related methods. We refer to the cookie, the role bitmap, and the resources associated with a cookie as a **role**. A cookie is essentially a **capability** [105, 11].

During start-up of AOS, an *init role* is generated, that may invoke any method on the AOS API. The init role is used by an *init process* that controls usage of AOS (think of *init*

process in UNIX). Given the init role, the init process can generate other roles, called *child roles*, for middleware processes that request a role. Generally, a role created by the init process allows the middleware to create additional subroles, e.g., to be used by components within the middleware that manage a particular task. Upon role creation, AOS verifies that the bitmap for the child role does not exceed the creator's role bitmap. Sibling roles (roles created by the same role) cannot access each other's resources.

Roles determine ownership of resources. Resources include agent containers, communication channels, and child roles. Child roles are owned by the creating role. This way, the init process can control (and delete) all AOS resources; effectively, AOS roles form a tree. Child roles (and their subroles and associated resources) can be deleted by their creating role only. When a role is deleted (using an AOS call), all resources associated with this role are deleted, including subroles and their resources. All roles are stored in an internal AOS table, together with an (initially empty) list of resources owned (created) by each role. Using roles, AOS can verify whether a method invocation is allowed and whether the resource that is referred to is owned by the invoking role.

4.4.4. Middleware compartmentalisation and security

An interesting challenge is to determine how roles are assigned to different processes in a middleware system. In a simple implementation, all processes within a middleware can have access to the same cookie, and thus to all AOS resources of this middleware. Another example is where a service may only use communication related calls and resources, and agent servers may only access the ACs of the agents they manage. Assigning a different role to every middleware component allows for compartmentalisation of the middleware system. This avoids a situation in which a single compromised middleware component can compromise the state of other middleware components, for example, by destroying AOS kernel objects such as ACs.

Roles can, if required, be passed between processes. As an example of using roles, consider a central middleware process that receives incoming ACs using AOS, and then dispatches these to appropriate agent servers. This central middleware process creates a new role before invoking the AOS call to receive an AC. The received AC is now owned by, and only accessible to, this role. At this point, the middleware process can inspect the AC's content and enforce a central access control policy. If the agent is permitted to enter, the central middleware process passes the role's cookie to an agent server process which can then retrieve the agent's code segment from the AC and start the agent. An agent server can only access ACs for which it was given a role, and only takes care of life cycle management; in this example, it is not allowed to receive new ACs itself.

The role model specifically allows construction of modular middleware that adheres to the principle of least privilege. Compartmentalisation avoids that a single compromised middleware component can exceed its privilege (as it has a role which only allows the minimum

required operations), and prevents compromise of resources owned by different middleware components, such as ACs and communication channels. As illustrated, different compartmentalisation strategies can be implemented.

4.5. Implementation

This section provides an overview of the AOS implementation, and the AOS API. AOS is implemented using a number of internal modules; agent containers, the agent container transfer protocol (ACTP), and communication.

4.5.1. Internals of AOS and RPC dispatchers

AOS supports multiple so-called **RPC dispatchers** for different RPC implementations. Multiple dispatchers can run simultaneously, so that middleware components written in different languages can use different RPC interfaces. Each dispatcher can serve multiple clients concurrently.

AOS consists of the main AOS kernel, which implements the AOS interface. Linked in with the main AOS kernel are one or more dispatchers which provide an RPC interface using which the AOS methods can be called. Dispatchers implement a specific RPC interface, using which a middleware system can invoke AOS calls (Sec. 4.3).

In the current implementation, RPC dispatchers have been written in XML-RPC, Sun-RPC, and Java-RMI. The SunRPC and Java-RMI dispatchers are currently used in the C kernel and the Java implementation of the AOS kernel, respectively. An XML-RPC dispatcher is implemented for both kernels. All dispatchers are reached on a different communication endpoint. Each dispatcher writes its endpoint into a file (in a directory ~/.aos/) in the home directory of the user whom started AOS. Middleware processes are linked with a stub library that is initialised with this endpoint (usually, a TCP port). Combined with a cookie, the middleware can invoke methods on the AOS kernel. Every dispatcher has at least one thread that waits for incoming connections or RPC requests.

As an example, the SunRPC dispatcher implementation written for the C++ kernel, is reachable over UDP and TCP. One thread listens to incoming TCP connections. Another thread handles incoming requests over UDP. Another TCP-based dispatcher implementation can create a thread per connection to handle incoming requests. The dispatcher threads for handling RPC requests are decoupled from the main implementation of the AOS kernel.

Fig. 8 shows multiple dispatchers on different kernels that serve requests from middleware processes.

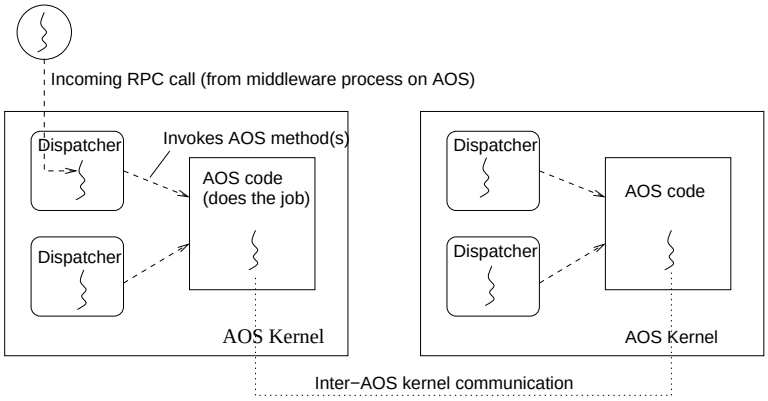

Fig. 8. RPC dispatchers. Two AOS kernels are shown, each having two dispatchers. Each dispatcher handles invocations for one or more processes. Each dispatcher has at least one thread waiting for RPC invocations, or it might have multiple threads (e.g., one thread per invocation).

The AOS kernel internally uses only two main threads. One thread handles incoming connections and incoming data over (multiplexed) communication channels, another thread handles incoming ACTP requests (for *ship/wait_ac*, see AOS API below). These two threads correspond to two internal protocols defined in AOS. One protocol is defined for multiplexed communication, the other for agent container transport. The two threads are not visible to the RPC dispatchers. The AOS kernel provides a native AOS interface to dispatchers, which corresponds to the AOS API discussed in the following section.

The C++ kernel internally contains a number of modules. Each module handles specific functionality of the AOS kernel. Specifically, modules exist for handling agent containers, (ScID-based) authenticated and encrypted SSL (over TCP/IP) ‘‘base channels’’ which are used for communication between AOS kernels. The multiplexed communication protocol (MUXP) and the agent container transport protocol (ACTP) are layered on top of these base channels. All AOS modules implement thread safety using condition variables and mutexes where necessary, to prevent problems that may occur due to concurrent access to AOS methods. This way, the AOS kernel can support the use of multiple RPC dispatchers that handle multiple incoming RPC requests concurrently.

Note that it is theoretically possible to implement the AOS kernel as a library that can be linked in with a middleware system. Assuming that the middleware and the AOS kernel are implemented in the same language, doing so could even be straightforward, depending on the language that the middleware is written in. Such a ‘‘runtime AOS kernel’’ could be a performance optimisation for middleware systems in which using a separate AOS kernel per process (each requiring its own TCP port) is not a problem, but where performance is.

4.6. The AOS API

Fig. 9 gives an overview of the AOS interface. The interface consists of methods for managing roles, methods for managing AC's and AC transport, and of methods for managing communication endpoints, including a socket-like interface for communication. A full description of the AOS interface can be found in a separate document which contains the AOS design notes [81].

Create_role is the method used to create roles, as described in Sec. 4.4.3. The init role's *cookie* is known to the init process, and can be used to create the first child role(s). A **cookie** is a 128-bit random number generated by AOS; the init role's *cookie* is created when AOS is first started up. *Role_bitmap* is a 32-bit bitmap, with a bit for every method of the AOS API, with the first method as the leftmost bit. A 1 bit means invoking the method is allowed, a 0 bit means invoking the method is not allowed. The *create_role* checks that the child role does not have any bits on which were 0 in the parent's role; if this check fails, an error is returned. *Delete_role* deletes a child role and its subroles, including associated resources.

Create_ac allows for creation of a new agent container. *Delete_ac* is used to delete an existing one. *Create_seg* is used to create a new segment in an AC. *Type* indicates one of a set of basic types defined by AOS, such as *TOC, SIGNATURE, CODE, LIBRARY, AGT_DATA, AOS_SYS* and *APP_SYS*. *APP_SYS* is used by the middleware, *AGT_DATA* is used to store data on behalf of an agent. *Subtype* can be used to associate additional subtype to the above types. It can be any middleware-defined string. An example is a *linux_x86* subtype with a *CODE* segment. *Descr* is a middleware-defined name, typically just a name for the segment. The *read/write/delete_seg* calls do what they indicate. *Write_seg* and *read_seg* allow for writing data to segments, or reading from them. An *offset* is specified; this defines the place where the read or write should start. An *offset* beyond EOS (end of segment) is not allowed (holes are not allowed).

Method (arguments)	Description
create_role(cookie, role_bitmap, *childrole)	Create a child role
delete_role(cookie, childrole)	Delete child role/cookie
create_ac(cookie)	Returns ACID
delete_ac(cookie, ACID)	Delete AC
read_toc(cookie, ACID, offset, n, *tocent)	Read table of content entries
create_seg(cookie, ACID, type, subtype, descr)	Create segment with type/subtype/name
read_seg(cookie, ACID, seg_id, offset, n, *buf)	Read data from segment
write_seg(cookie, ACID, seg_id, offset, n, *buf)	Write to segment
delete_seg(cookie, ACID, seg_id)	Delete segment
make_persistent(cookie, ACID, seg_id)	Make persistent (irrevocable)
finalize_ac(cookie, ACID)	Finalize the AC
ship_ac(cookie, ACID, endp, xid);	Ship AC to waiting middleware
prepare_wait_ac(cookie, suites, *actp_ep, *xid)	Endpoint for ACTP
wait_ac(cookie, *peer, *xid, block_time);	Wait for XID (0 for any XID)
create_listen_endp(cookie, index, suites, *endp)	Create listen endpoint; returns descriptor
delete_listen_endp(cookie, port)	Delete endpoint
accept(cookie, descr, *peer_mcr, block)	Wait for connection
connect(cookie, *mcr, suites)	Connect. Returns descr
send/recv/peek/close (cookie, descr, len, *buf)	Data send/read and close connection
get_parm(cookie, name);	Obtain compile-time/configured limits
reenable_role(cookie)	Reenable role after AOS restart

Fig. 9. Overview of the AOS interface

The *make_persistent* call sets a bit in the table of content entry for the indicated segment, which turns the segment into a persistent segment. Initially, all segments are transient. After setting the persistent bit, the segment becomes immutable. This bit is used by the audit trail mechanism described later in this chapter, to check whether any illegitimate changes have been made to persistent segments later in the agent's itinerary. Persistent segments help agents ensure that certain data (e.g., an offering for a product) ends up with the agent owner in unmodified form.

Finalize creates (updates) checksums of all segments, SHA-1 hashes, in the ToC entries of all segments in the table of content, signs the ToC, and syncs all segments to of the AC to disk for crash recovery. *finalize* must be called before calling *ship_ac* to migrate an AC, because a finalized and signed (by AOS) AC is needed for integrity protection when receiving an AC. The latest *finalize*'d ToC is stored in a known location in the AC (segment 0). The signature and the public key of the signing AOS kernel can be found in segments 1 and 2, respectively.

Prepare_wait_ac provides a way for a middleware process to create an endpoint (in AOS) for AC's. The endpoint consists of an AOS **agent container transfer protocol (ACTP)** endpoint *actp_ep*, combined with a unique 128-bit transaction identifier (*XID*). The AOS endpoint is a normal AOS kernel contact record, as shown in Fig. 10, with the *index* field set to to 0. The AOS endpoint may be shared for different AC transport transactions; *XID* is used to disambiguate different agent shipments; it is a 128-bit number to ensure that it is hard to guess. The protocol is described in detail in Sec. 8.3.2.

The middleware[20] may wait for different AC's (different XIDs) at the same time using a single *wait_ac* call. If the middleware, or a thread in the middleware, listens for a specific AC, it specifies the XID. If it waits for *any* XID, it specifies a zeroed XID. The *wait_ac* call then fills in the XID of the first incoming AC.

*XID*s are coupled to roles; *wait_ac* will only return for AC's with a XID that is registered with the role that invokes the *wait_ac* call.

A middleware system can register an AOS-level agent container transfer protocol (ACTP) endpoint in a location service. It is however advisable to implement an *end to end* handoff protocol at a higher level (Sec. 8.3.2). Mansion, for example, registers a middleware-level agent transfer protocol (ATP) endpoint in its location service. This middleware waits for connections from another middleware process. After a connection is made and agreement is reached on whether an agent may be shipped, the receiving middleware creates an ACTP endpoint on the AOS kernel obtained using the *prepare_wait_ac* call, and returns that endpoint including the XID to the sending middleware, which can then invoke the *ship_ac* call to ship an AC to this endpoint. Assuming an end-to-end authenticated channel to exchange the ACTP contact record and XID between receiving and sending middleware, this procedure ensures that an authentic AOS ACTP contact record is used to ship the AC.

The AOS communication interface is quite straightforward. It is similar to BSD sockets except for the way in which communication endpoints are created.

Create_listen_endpoint is used by a middleware-level process to create an AOS endpoint. In this case, the *index* field in the AOS endpoint is used, indicating the specific endpoint that has been created. The call returns an AOS contact record describing the endpoint in a caller-provided buffer; the AOS contact record is explained below.

Delete_listen_endpoint can be used to delete the endpoint. *send* and *recv* calls send and receive data from and to a buffer in the sender's address space. Bytes are received in the same order as they are sent; the channel is ordered and reliable.

Connect and *accept* are used to establish a connection. These calls return a descriptor that can be used by *send* and *recv*. *Close* closes the descriptor and the channel.

Peek is identical to *recv*, however, the data are not removed from the buffer internal to AOS. A subsequent read reads the same bytes from the stream as read before with peek.

Select takes a list of connection identifiers (*descriptors*), and returns their status (readable, writeable, exception). The semantics of select and its descriptor sets is identical to its counterpart in BSD sockets (see [108]).

[20] In Mansion, this would be the MMW process.

Get_parm is used to obtain compile-time parameters or limits from AOS, such as the maximum segment size or the maximum size of an AC.

Finally, *reenable_role* is a method used for crash recovery. If an AOS kernel crashes or is killed, and is used again, invoking one of its methods returns an error to indicate that it has crashed. In this case, all resources (such as communication endpoints) are lost, but finalized AC's can be recovered. After *reenable_role*, AC resources of a given role again become accessible.

4.6.1. The AOS endpoint record

The *AOS endpoint* data structure (Fig. 10) is quite straightforward. It contains an IP address (IPv6, or an IPv4 address in IPv6 format), and a TCP *port*. This indicates the TCP endpoint on which the AOS kernel listens for incoming connections. Further, the endpoint contains a Self-certifying ID *ScID*, and an *index*. Note that AOS is part of a communication stack, and that its AOS endpoint may be embedded in a middleware endpoint record.

```
typedef struct aos_endp {
  char ip-addr[16];
  short port;
  char scid[20];
  int index;
} aos_endp_t;
```

Fig. 10. AOS endpoint data structure

The *index* field indicates an endpoint relative to AOS that other processes can connect to (created using *create_listen_endp*), or an endpoint that an agent container can be sent to. The ScID is in binary format, not base32 encoded, so it is 20 bytes.

The *scid* in the AOS endpoint is the hash over the public key of the AOS kernel. AOS kernels always authenticate their peer. The *connect* call takes an *aos_endp* as an argument; if the ScID is filled in before connecting, the local AOS checks that the remove AOS kernel has the private key corresponding to the ScID. If not, the connection fails. If the ScID is not filled in before connecting, it is filled in by the AOS kernel as a result of the *connect* call. The same applies to the accept call.

4.6.2. Usability of AOS

AOS was designed to be used in the AgentScape and Mansion agent systems. This section evaluates the usability of AOS in Mansion.

One of the reasons for using a layered approach is that certain (common) functionality can be delegated to a lower layer in the stack. The main reason for choosing AOS as a layer in

Mansion is that it can implement the agent container abstraction and AC transport mechanisms, such that Mansion does not have to implement this. Using the AC management and transport mechanisms is very convenient.

For communication, the main advantage of using AOS is that a single TCP port can be used to accept connections for multiple processes on the same machine, for example, behind a firewall. Another reason is that AOS can implement the mechanism for setting up authenticated encrypted connections (using a ScID-based authentication protocol) for the middleware, and thus that the middleware implementation can be simplified. For Mansion, this advantage is limited as it needs to implement various security mechanisms, in particular end-to-end authentication, anyway.

There are advantages to having confidentiality (encryption) implemented in AOS instead of the middleware layer. For one, setting up authenticated, encrypted connections is computationally expensive. If multiple AOS can establish underlying encrypted ''base channels'' over which connections from multiple independent processes that use the same pair of AOS kernels, are multiplexed. This allows reuse of (possibly persistent) connections between two AOS kernels by multiple processes that use these AOS kernels, amortising the cost of connection setup, establishing symmetric key material, etc. On the other hand communication through AOS comes at the cost of increased latency due to the IPC overhead imposed by communicating through AOS. For this reason, the communication system of Mansion (Chapter 5) allows users to choose TCP/IP as the underlying substrate instead. Also, Mansion makes sure not to use AOS for local communication; if the Mansion communication layer detects connections to a port on the same machine, it will connect directly irrespective of the chosen ''substrate.''

How AOS is used to construct primitives in the Mansion middleware stack will be discussed in Chapter 5. The remainder of this chapter will focus on AOS performance.

4.7. Performance of basic AOS primitives

For a central component such as AOS, used for all interprocess communication, mobile agent code/data management and migration operations, performance is highly important. This section presents the approach and performance of two AOS kernels implemented and used in our department. In particular, communication throughput and scalability, and AC shipment related overhead and scalability in terms of concurrently shipped ACs.

Independently, two versions of AOS have been implemented, one in Java and one in C++, based on a precise specification of the AOS interface and the internal protocols used by AOS. Both the Java and the C++ kernel are used to construct two different agent middleware systems, Mansion [79] (written in C) and AgentScape [127] (written in Java). These two mobile agent systems differ in their design and implementation decisions; even so, AOS has shown to be a solid basis for their construction.

This section evaluates the performance of the Java and C++ AOS kernels. The tests in this section were run on a dedicated 1 GHz dual Pentium-III machine with 1 GB memory, running Linux on an ext3 file system and using a Fast Ethernet (100 Mbit/s) local area network. End-to-end tests described in a later section are run on a more recent machine. Tests with the Java kernel used the Sun Java 15 standard compiler and Java HotSpot server virtual machine version 15. The cryptographic libraries used in the Java AOS kernel are from Bouncy Castle[21].

The tests are run with modified AOS kernels that included microsecond timers, and are executed 5 to 10 times in a row, with averages shown in this section. For all tests that use AOS-to-AOS communication, the connection is configured to use 128 bits AES encryption with SHA-1 message authentication. AES provides a reasonable trade-off between security and efficiency, compared to, for example, 3DES.

4.7.1. AOS-to-AOS communication cost

AOS uses an internal protocol to multiplex communication channels over a single internal encrypted ‘‘base channel’’ Figure 11 shows the performance and scalability of AOS for communication, for 1 to 16 threads communicating concurrently over AOS.

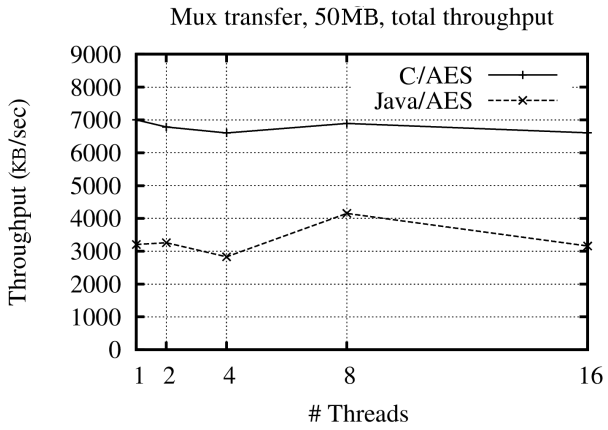

Fig. 11. Total throughput for multiplexed communication over a shared AOS-to-AOS connection.

In this experiment, each thread sends 25 MB over an AES encrypted base channel to a server process running on a different AOS kernel.

As shown in Fig 11, the C++ kernel has a substantially higher throughput than the Java kernel. We attribute this to the fact that the OpenSSL library implemented in C is faster than the (JIT compiled) pure Java Bouncy Castle SSL implementation used in the Java kernel[22].

[21] http://wwwbouncycastle.org

[22] Performance measures with an unencrypted (NULL) SSL channel show that Java performance in the unencrypted case comes close to the performance of the C++ kernel in the same scenario. Please note that the HotSpot JIT compiler uses *eventual*

Both kernels apply locking strategies to make sure that only a single thread can write payload on the base channel.

The figure shows that overall throughput differs between the C and the Java kernel, but that the total throughput stays roughly the same for both kernels, irrespective of the number of threads that simultaneously send payload over the wire, although some variation exists which remains unexplained. Although the per-thread throughput decreases linearly with the number of threads for obvious reasons (i.e., sharing and overall saturation of the underlying connection), the measurements show that the internal protocol used for multiplexing in AOS do not adversely influence scalability.

4.7.2. Finalize costs

Prior to shipping an AC, an AC is finalized to ensure that the AC's table of content is generated, and that all segments are stored in a zip file synchronised to disk. *Finalize* is a call that constructs a secure ToC of the AC and signs it, prior to shipping it to another AOS kernel In addition, *finalize sync*s the AC to disk for crash recovery reasons.

Table 12 shows a micro-benchmark of the finalize costs of agent containers of 500 KB, 1 MB and 5 MB containing random data. These sizes are typical for many agents used in our own agent middleware system. ToC checksumming and signing cause little overhead, also for large ACs, and this increases linearly with the size of the AC; the checksum (SHA-1 hash) generation takes place over every byte of every segment.

	C++			Java		
	500KB	**1MB**	**5MB**	**500KB**	**1MB**	**5MB**
checksum	9	19	98	36	74	70
sign	51	52	70	5	16	51
zip	133	248	1356	145	303	1449
sync	166	238	922	179	401	1623
total	359	558	2446	442	878	3854

Fig. 12. Finalize micro-benchmarks (in milliseconds) for resp. the C++ kernel and the Java kernel.

Creating a zip file and synchronising it to disk requires that each segment is copied into the zip file. The resulting zip file is synchronises to disk. Table 12 shows both steps cause substantial overhead, particularly for large ACs. Finalize times scale roughly linearly with the AC sizes for both the Java and the C kernel, although finalize takes substantially longer on the Java kernel than on the C kernel.

As mobile agents may migrate often during their lifetime, AC finalization and transfer cost can increase the time for an agent to achieve its task considerably, and may influence

JIT compilation of frequently used bytecode. Furthermore, note that JIT compilation efficiency and optimisations may have improved since the time of these tests.

scalability of the mobile agent middleware as a whole. A straightforward optimisation for performance, is to have AOS ship segment files to another AOS kernel directly, without zipping the files first, in an FTP-like manner. Another straightforward optimisation for performance, is to let go of the crash recovery assurance by means of the *fsync* system call.

4.7.3. AC shipment cost

AC shipment is composed of a *ship_ac* primitive combined with a *wait_ac* primitive at the receiving end, that returns after shipment is completed. *Ship_ac* takes a finalized agent container, and ships it over an SSL connection. The receiving AOS kernel extracts the agent's zip file containing the agent's segments, and verifies the checksums in and the signature over its ToC. Only after this verification, wait_ac returns. After an acknowledgment is received for all shipped ACs, the timer is stopped. The *ship_ac* cost measured thus includes on-the-wire time and the extract/verify cost at the receiving end.

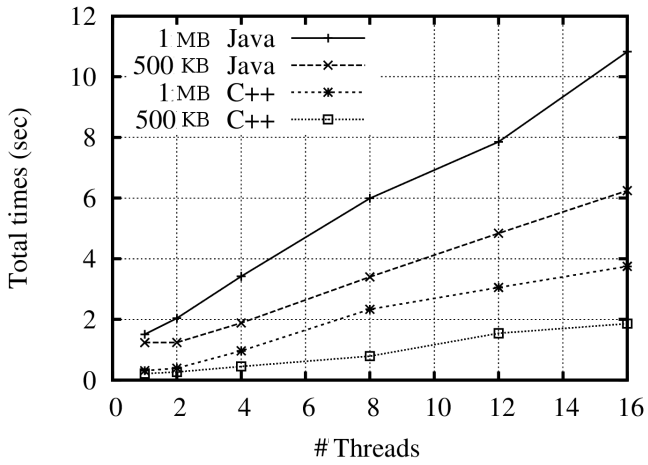

Fig. 13. Elapsed time to ship 1-16 agent containers of 500 KB resp. 1 MB for both the Java and the C++ kernel.

The total *ship_ac* costs including AC extraction and verification are measured for AC sizes of 500 KB, 1 MB, and 5 MB containing identical segments of 5 KB random binary data. The cost of extracting and verifying an AC after it is received depends primarily on the size of the AC. The times for zip extraction (default compression ratio), signature, and checksum verification in the C++ kernel are 0.064, 0.127, and 0.734 sec. in total for 500 KB, 1 MB, and 5 MB, respectively. Extraction, signature and checksum verification in the Java kernel takes substantially longer, namely 0.597, 1.033, and 3.472 sec. in total, respectively. Of these times, about 80-90% is spent on unzipping the AC.

Figures 13 and 14 shows the results for both the C++ kernel and the Java kernel for 1, 2, 4, 8 and 16 *ship_ac* calls at the same time. The figures show that AOS *ship_ac* calls scale roughly linearly with concurrent use. The figures also show that the time needed to ship an AC is more for the Java kernel than for the C++ kernel. This can be attributed in part to the fact that cryptography (for encrypting the connection) and AC extraction and verification take longer in Java than in C++.

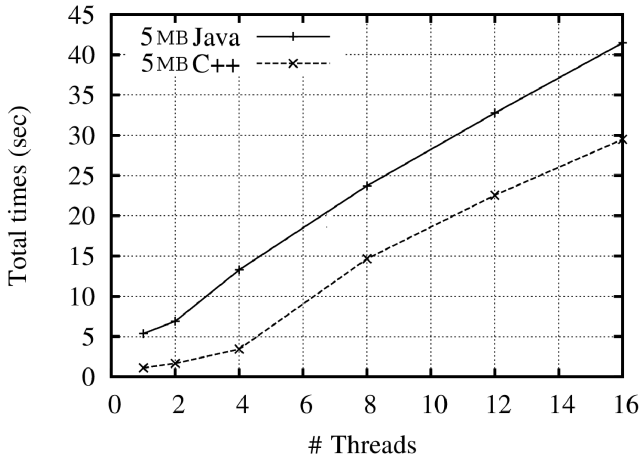

Fig. 14. Elapsed time to ship 1-16 agent containers of 5 MB for both the Java and the C++ kernel.

4.8. Related work

Existing mobile agent systems are designed with different goals and foci, for example, on communication, mobility, security, agent model support, management, etc. [69]. Most existing agent platforms are implemented as monolithic systems, where all functionality is integrated in a single code-base. Interoperability specifications like FIPA or MASIF define higher-level functionality such as inter-agent communication protocols. These specifications do not define low-level protocols for, for example, agent migration or for communication within middleware systems. As AOS is not an agent middleware itself, but rather a middleware building block, comparing AOS with full functional middleware can only be done partially by considering the leading design requirements.

The FIPA standard specification[23] includes a series of documents describing the functionality and operation of agent middleware FIPA compliant agent middleware can interoperate which each other, e.g., agents can exchange messages, interact with, and reason about agents on other middleware. One of the most widely used FIPA compliant agent middleware is JADE [20]. The latest middleware design (version 35 as of today) is modular in design and many parties (universities and companies) have contributed to JADE. The middleware is implemented in Java and supports a Java API for agent development. It is a complete self-

[23] http://www.fipa.org/

relying system, with integrated location and yellow pages services. This is different from the AOS perspective to agent middleware, where services reside at the middleware level.

Ajanta [55] is designed to include a number of security primitives and architecture features to protect both the host and the agent from malicious actions. It includes amongst others a similar concept as the agent container in AOS, allowing for an audit trail mechanism resembling the one outlined in this paper and in [79]. However, Ajanta is completely Java-based and is not designed to incorporate or interact with other software components or services.

The Tacoma [51] project focuses on operating system support for mobile agents. In that respect, it has many similar design goals as AOS by providing abstractions for, in particular, data storage and agent mobility. Although it also provides a simple container abstraction, called a ''briefcase,'' only simple protection mechanisms were implemented. Tacoma supports multiple programming languages for agents, including C and Tcl/Tk.

The MadKit agent platform architecture [47] aims to provide a generic multiagent platform. The architecture is based on a minimalist agent kernel decoupled from specific agency models Although there are similarities with the design goals of the architecture model with AOS, the design and implementation is quite different. The aim of MadKit is to allow a developer to implement his or her own agent architectures Basic services like message passing, migration, monitoring, or management, are provided by platform agents. MadKit comes with a set of ''containers,'' realising different execution environments for running an application.

Compared to other work, AOS is unique in its goals. In contrast to existing systems, AOS is not directly used by agents, but instead it aims to provide a generic, secure layer that us usable for constructing different agent middleware systems.

4.9. Conclusion

This chapter discusses the design requirements, implementation, and performance of the AOS kernel. AOS is a portable middleware building block specifically aimed at constructing mobile agent middleware systems. It can be used by different middleware processes, possibly of different users, where each such process may be implemented in a different language. Programming language flexibility is facilitated by different RPC dispatchers, each providing a method invocation interface suitable for a specific language. The AOS design allows for secure sharing of a single AOS kernel between different middleware processes: it provides effective software fault isolation and safety by separating resources created by different middleware processes.

AOS provides a minimal set of primitives for mobile agent systems, in particular for agent code and data storage, agent transport, and for communication between mobile agent middleware components. AOS provides basic security services which can be used by higher-level middleware layers to construct more elaborate security, such as authentication mechanisms and secure agent transport and auditing of mobile agents. AOS does not impose a specific model on the agent middleware. AOS is used in two different agent systems, illustrating

that AOS provides a level of abstraction that can be used to construct diverse mobile agent systems with—even if those have rather different requirements or designs.

Two implementations of AOS have been built and tested for interoperability. Performance measurements of the AOS kernel are shown in this chapter. These show that AOS performance differs between the Java and the C++ kernel. The C++ kernel outperforms the Java kernel for most tests, primarily due to the fact that C is more efficient than Java for tasks such as cryptography, which is used throughout the AOS kernel. On the other hand, Java provides better portability, and the Java kernel has been used to run the AgentScape mobile agent platform on Linux, Solaris, Mac OS X, and Windows systems The C++ kernel is currently only available for Linux and Solaris platforms.

Both implementations of AOS are shown to scale well with respect to concurrent usage by middleware systems for communication and transport of agent containers, which is important when using AOS to construct mobile agent systems that operate under load. Most of the time for agent container transport is spent on zip file construction as part of the finalize call, which is needed for AC transport. This can be optimized away by shipping segments directly instead of in a zip file. Other operations (e.g., hashing files for ToC construction) are quite efficient.

AOS offers a flexible basis for the construction of secure mobile agent systems, and for deploying multiple services or middleware processes at the same time on a single AOS kernel. Support for middleware that *internally* consists of components written in different languages is a novel contribution of our work. The access control model based on roles allows for applying the principle of least privilege within a modularly designed agent system, and offers the required separation of resources in scenarios where AOS is shared between different mobile agent systems or middleware components.

.

Chapter 5
Communication Layers and RPC

This chapter describes the system from a layering perspective: all middleware components make use of the same communication system, which consists of an RPC system layered on top of a secure communication layer. This chapter describes the layers, their interfaces, and how these are used.

The Mansion middleware internally consists of several components and services. These include the Middleware process (MMW) hosting agents, the Mansion Object Server (MOS), and services such as the location service and the basement, which are typically implemented as objects that run in a MOS. Mansion views the operating system (and the Network) as the lowest layer of the middleware stack. Mansion relies on the local operating system to provide a process execution abstraction, and network communication primitives. Every module, service, or process in Mansion uses the same communication system, layered on top of the operating system primitives. These communication layers provide the basis for secure (wide area) communication, RPC, and for agent transport.

The reason for designing a layered system is simple: all middleware processes share functionality, and it is useful to implement common functionality in a layer such that different components can reuse this functionality. This chapter describes the layered communication system and the Mansion location service needed to make use of it.

5.0.1. Requirements

Security, particularly authentication, confidentiality and integrity are important design criteria for the communication layer. Encryption of transport is a basic requirement to ensure confidentiality of interagent communication, object method invocation, and agent transport. Some of these interactions require wide-area communication. Because of the middleware's modular design, request/reply (RPC) communication patterns are common. Using a form of secure

RPC would be convenient.

Symmetric key cryptography is much more efficient than public key cryptography (see [36]), A connection-oriented transport like SSL/TLS allows setting up authenticated, secure channels that use symmetric key cryptography to encrypt data which is sent over the connection efficiently. SSL requires public key cryptography during authentication and for secure exchange of symmetric keys in the initial authenticated key exchange (handshake) phase. Once symmetric key material has been set up, the overhead of the initial public key-based handshake can be amortised over the time that the connection is used. Symmetric key cryptography connections are much more efficient than a message oriented communication system where every message has to be encrypted individually [56]; this would also apply to encoding RPC messages.

Mansion (zones) cannot assume a local, trusted network. Zone member processes may reside on hosts that are located far apart. Thus, where some (fast) RPC implementations used efficient asynchronous datagram communication, it is useful to bias Mansion RPC towards security instead. In addition, if there is no need, it would be useful not to assume fixed side (RPC) messages in Mansion. Message lengths may differ in Mansion—for example, the communication communication strategy of agents cannot be predicted; agents may send collected data to each other. Internally in Mansion, this may be represented by a single RPC call pushing data to another agent's middleware system. RPC message sizes may thus range from fixed-sized records returned by a location service to variable-length data streams resulting from communicating with an agent or invoking an object.

As security is a requirement in all cases and wide-area communication is common, a secure channel abstraction was chosen as the basis for communication and RPC.

5.1. Layering

Mansion uses reliable connections as the underlying abstraction for communication. The basis is TCP/IP. On top of that, Mansion implements a (zone-based) authenticated transport layer based on SSL/TLS. A **remote procedure call (RPC)** system is layered on top of that.

Mansion processes use RPC to communicate with other Mansion processes. RPC provides a synchronous communication mechanism based on request-response messages. RPC is convenient: processes provide (define) an interface that can be called by other processes. A protocol (or interface) designer does not have to worry about defining messages or deal with communication issues. An RPC compiler generates RPC code (e.g., in C or C++) which marshalls an invocation into a request message, ships it to a remote service, and after the response arrived, unmarshalls the reply and returns it to the caller.

The RPC layer is based on an underlying secure socket abstraction. These sockets can in principle also be used directly by middleware processes. Irrespective of whether communication channels are used directly or through the RPC layer, all connections in Mansion are authenticated using self-certifying identifiers (Sec. 3.2.1).

The Mansion communication and RPC layers are the basis for agent transport abstractions, object method invocation, and interagent communication primitives. These are the layers in the Mansion communication stack:

1) The **network layer.**[24] The network is part of the operating system. Availability of TCP/IP (both IPv4 and IPv6 are supported) on the local host is assumed.
2) On top of TCP/IP, a portable secure socket abstraction is constructed, with a BSD TCP-socket like interface. This layer is called the **ZAC layer,** for **zone authenticated communication** layer. ZAC communication channels are reliable, ordered, bidirectional, authenticated channels which protect confidentiality. The ZAC layer uses SSL internally, but does not expose X.509 public key certificates to the application and does not require use of an X.509-based public key infrastructure.
3) On top of ZAC is the **RPC layer,** or remote procedure call layer. Our RPC system is similar to the system devised by Birrell et al. [27], except that it uses authenticated, encrypted connections provided by ZAC.
4) The **middleware** and the **application layer** are not strictly part of the communication stack. However, these layers are important for issues such as key management and end-to-end authentication. Fig. 15 shows the layers described above.

All middleware processes in Mansion are a member of a zone. The ScID-based zone authentication protocol described in Sec. 3.2.1 is used to implement authenticated, encrypted ZAC connections.

Fig. 15 shows a figure of the layers used in Mansion, using a MMW process with agents, an object server with objects, and a Mansion (internal) service such as a location service, as examples. It also shows AOS as an (optional) layer used by ZAC. Fig. 15 shows three different middleware processes: the Mansion middleware (MMW), the MOS, and a service, on three hosts. The RPC and ZAC layers are implemented as libraries which are linked in with the middleware binaries. The AOS layer is implemented as a separate process that cannot be linked in. Communication between ZAC and AOS takes place using SunRPC or XML-RPC (see chapter 4).

Mansion's internal structure is modular. Separate components of the middleware are typically implemented as separate processes. These components each handle certain tasks, like managing objects, or providing contact information of other middleware components. Middleware processes communicate with each other using RPC calls. If the processes reside on different hosts, RPC takes place over authenticated, encrypted communication channels provided by the ZAC layer. For local communication, an unencrypted TCP connection is used in ZAC, although processes must still authenticate.

[24] Not to be confused with the OSI network layer.

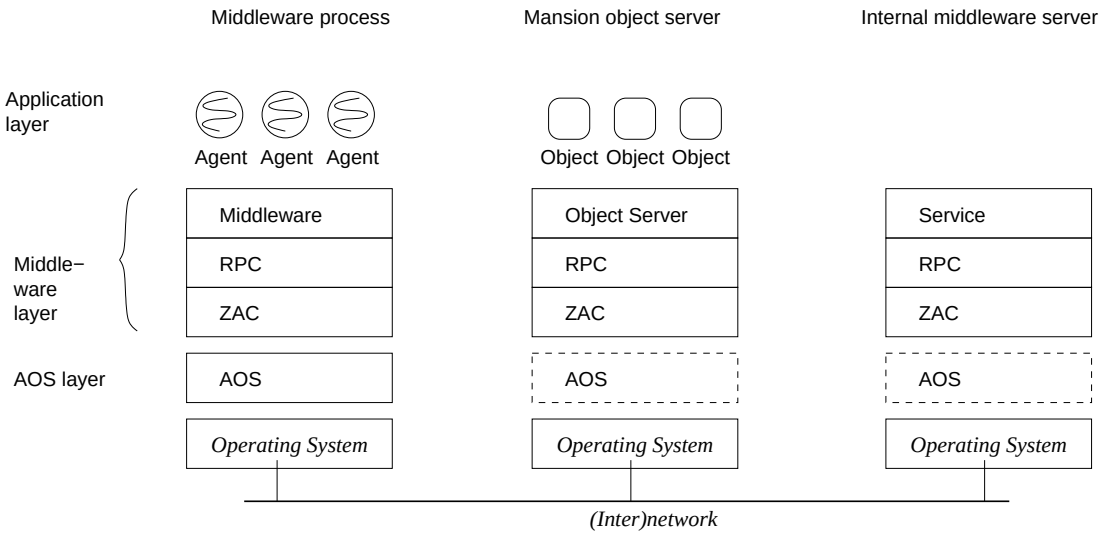

Fig. 15. Middleware layering for three different examples of middleware process (on different machines). Layers in Italic font (OS and network) are not part of our software, but all Mansion middleware components depend on them. The AOS layer (dashed) is optional, except for the MMW process which needs it to manage agent containers (ACs). Layers above the operating system are described in the text.

This chapter explains the ZAC layer and the RPC layer in detail, beginning with a bottom-up explanation of the layers shown in Fig. 15. Combined, this chapter gives a complete overview of the communication infrastructure used in Mansion.

5.1.1. ScID-based authentication

This section describes the ScID-based authentication protocol, using which two connecting processes authenticate each other. It is implemented in the ZAC layer.

All processes in Mansion have a public/private key pair. Every zone member process has a public key certificate signed by the zone owner which indicates that the holder of the public/private key pair is a member of the zone. The *ZoneID* is the hash over the public key of the zone owner. (Sec. 3.2.1). If a process is *not* a member of a zone, it has a self-signed certificate, the hash of which is the ScID.

ZAC uses the RSA authenticated key exchange protocol. As part of this, processes present their public key certificate. If the process is part of a zone, it must also present its certificate chain up to the zone certificate (i.e., the certificate of the zone owner).

SSL is used to implement the RSA authenticated key exchange protocol to set up an authenticated, encrypted connection. The symmetric key algorithm used for encrypting data after authentication and the message authentication code (MAC) algorithm can be configured by the user. Examples are 3DES and AES. The default is AES with SHA-1. A set ‘‘cipher

suites'' strings can be defined in the world design document (Sec. 3.8.1) to override the default.

SSL has been chosen for pragmatic reasons: well-engineered, tested implementations of SSL exist, including tools and libraries (we currently use the open source *openssl* SSL/TLS implementation and toolkit), whereas implementing an authenticated key exchange protocol from scratch is difficult and error-prone.

ZAC is not tied to SSL per-se; if a better toolkit or protocol than (open)SSL comes along, it can be replaced. The ZAC interface hides implementation details. None of the details are exposed to the application, except that the application program must be initialised with appropriate keys and self-signed (X.509) certificates. The ZAC interface knows about ScIDs only.

5.1.2. Implementation

SSL/TLS is designed around X.509 certificates. Normally in SSL, a public key certificate is signed by a CA which binds information about the owner of the private/public key pair to the public key. Information defined in X.509 certificates are, among other things, an organisation's name, or a *Common Name (CN)*, for example, a Web site's address [98].

Mansion uses SSL (x509) public/private key certificates for implementation, but these certificates are not exposed to the application. The communication API used by Mansion uses ScIDs. These correspond to self-signed (endpoint) certificates or self-signed zone certificates. These certificates are currently X.509 certificates, but could be replaced by another certificate container without changing the model; the X.509 PKI model is not used, nor are trusted certificate authorities such as used in the Web (e.g., *verisign*) needed. Mansion uses X.509 certificates only to store and exchange (signed) public keys. Except for keys and possibly signatures, nothing is stored in the certificates.

Zone-based authentication uses a few minor extensions to the existing openSSL library[25] These extensions are implemented in a library called *zonelib.* Zonelib is integrated in the ZAC layer. It customises the SSL certificate verification procedure (which assumes availability of a trusted CA list unused in Mansion). Zonelib retrieves the peer certificate(s) used for authentication of the peer process, and establishes the corresponding ScID by hashing the public key of the peer. In case of a zone member, the certificate chain (of length 2) is retrieved by SSL, and what is hashed is the root of this chain, the zone certificate.

5.1.3. Mansion contact records

Mansion has a standard contact record, which contains the information needed to connect to a middleware process. This contact record is used for all entities—middleware/MMW

[25] http://www.openssl.org/

processes, objects, agents, location service nodes—has a fixed format, and can be stored in a Mansion location service. It is called the **Mansion contact record (MCR)**.

The MCR is needed to resolve location independent-identifiers or *handles* of entities, such as a MMW process that is ready to receive agents, an object (replica) in a MOS, or the endpoint of an agent that can be connected to. The MCR is used by the ZAC layer to connect (and accept) connections.

The MCR can be partially or completely filled in before being passed to the ZAC communication interface to connect or accept a connection. The MCR has to contain at least an IP address and a port, and it may contain an AOS contact record if AOS is used. The MCR contain a ScIDs. If a ScID of the target process are not known and filled in into the MCR before connecting, it will be filled in after authenticating the peer. Note that in Mansion, the ScID of the peer is normally known before connecting, e.g., from a RoomID that is used when following a hyperlink.

The MCR is depicted in Fig. 16. Note that the syntax of struct definitions is slightly different than for regular C-style structs. The syntax of the the Mansion IDL will be described later in this chapter.

```
typedef struct scid {
  char[20] scid;
} scid_t;

typedef struct mcr {
  struct aos_endp aos_ep;
  scid_t peer_id;
  scid_t zone_id;
  int index;
} mcr_t;
```

Fig. 16. The Mansion Contact Record

The MCR includes the AOS contact record as indicated in Fig. 10. If AOS is not used, by convention, the *index* field in the AOS contact record is set to 0. *Aos_endp* contains the IP address (in IPv6 format), the TCP port and the ScID of the peer process or AOS kernel. The *index* field in the AOS endpoint indicates an endpoint relative to AOS, that is, an endpoint created by a middleware process. If *index* is set to 0, the IP address and port indicate the TCP endpoint of the middleware process.

Zone_id is the middleware process' ZoneID, if it is a member of a zone. If *zone_id* is filled in when connecting to an endpoint, it is verified by the ZAC layer and filled in after authentication of the peer as part of connection setup, if the peer has a zone certificate. If the peer only has a self-signed certificate, *peer_id* is filled in instead. *peer_id* is also filled in for zone member processes, so that individual zone member processes can be distinguished from each other[26]. Finally, the MCR *index* field indicates an endpoint relative to or in the middleware, e.g., an endpoint for a specific object or agent.

[26] This can be useful for tracking access or when zone member processes are blacklisted, for example.

5.1.4. The ZAC communication interface

The zone authenticated communication (ZAC) layer is a communication interface layered on top of AOS or plain TCP sockets. The ZAC interface provides the basis for all communication in Mansion, including RPC.

The ZAC interface contains primitives that closely resemble the BSD socket interface. It contains the following methods:

Method	Arguments
create_listen_endp(substrate, port, suite, *mcr)	Create an endpoint (MCR) in *mcr
accept(listen_descr, *peer_mcr)	Peer's authenticated MCR in peer_mcr
set_accept_callback(listen_descr, callback)	Register accept callback
connect(*target_mcr, suite)	Connect to target
accept(listen_descr, *peer_mcr)	Accept; peer MCR is returned
send(descr, buf, len)	Write len bytes from buf to channel
recv(descr, *buf, len, block)	(Blocking) read from channel to *buf
close(descr)	Close connection
select(*rd, *wr, *exc, block)	Poll/await connection status

Fig. 17. Overview of the zone authenticated communication (ZAC) layer interface

The interface in Fig. 17 is similar to standard BSD TCP/IP sockets, and has similar semantics. ZAC can use AOS or TCP as the underlying substrate. Whether AOS or TCP is used is determined by the process that creates the (listen) endpoint. In both cases, SSL/TLS is used to authenticate channels and set up the cryptographic material required to provide confidentiality. However if AOS is used, channel encryption for confidentiality takes place at the AOS level. Channels set up by the ZAC layer are always mutually authenticated using ScID-based authentication. Note that ‘‘*’’ is C-style notation for a pointer, used to denote call-by reference (out) arguments.

The *connect* and *accept* calls make use of the Mansion contact record described in Fig. 16. As explained above, when connecting to an endpoint, the AOS *index* field in the MCR allows ZAC to determine whether the connection should be established using AOS or directly over TCP/IP. *Accept* uses a descriptor that has been created earlier using the *create_listen_endp* call. The *substrate* argument of the *create_listen_endp* call determines the substrate to use: TCP/IP or AOS. When accepting an incoming connection, the substrate is implicitly clear through the *listen_descr* used for *accept*.

Create_listen_endp does not exist in BSD sockets, but resembles a combination of the *bind* and *listen* calls of BSD sockets. It creates an endpoint that other processes can connect to. The *port* argument specifies a specific TCP port to listen on when the *substrate* is TCP; if the substrate is AOS, it defines a specific *index* relative to AOS (AOS listens on a TCP endpoint allocated when it is started up). If unspecified, ZAC lets the underlying substrate

allocate a port. A *local* substrate can also be specified; this is useful to specifically create an endpoint for (RPC) communication between local processes on the same machine; here, a local TCP connection is used, with authentication but without subsequent encryption of data. The ZAC layer puts the resulting endpoint in the *struct mcr* argument; this MCR can be announced in, for example, a location service or distributed another way. The calling process obtains the endpoint's contact record after the endpoint is created, in a call-by-reference argument of *create_listen_endp*.

The *suite* argument is used to define a (set of) specific cipher suite(s) for connections to this port. The name is inherited from the SSL/TLS protocols; a cipher suite allows the caller to select a symmetric key cryptographic algorithm and a message authentication (MAC) algorithm used to encrypt the channel. Currently, the authenticated key exchange mechanism cannot be configured; this is always RSA. The *suite* argument is used by both the *create_listen_endpoint* and the connect calls, and a cipher suite name from the sets defined in both connecting and accepting party must match, else connection setup fails. If sec_suite is NULL, the ZAC layer (or underlying AOS substrate) uses the default cipher suite (AES-SHA1), or if applicable it selects a cipher suite from a the list of cipher suites defined in the world design document. Mansion by default uses an RSA-based authenticated key-exchange algorithm to establish an AES256 symmetric key. A SHA-1 MAC'ing algorithm is used for integrity protection and authentication of data.

Accept and *connect* take arguments of type *mcr*. These contain the Mansion contact record of the peer. By passing these arguments call-by-reference, initially missing values (e.g., AOS ScID or peer_ids) in the struct can be filled in by the ZAC layer.

An accept callback function can be registered using the method *set_accept_callback*. If registered, this callback is called at connection time, with mcr of the peer filled in as determined during authentication, before turning control to the *accept* call. The accept callback function may return 0 or 1, depending on a recipient's decision on whether to accept or deny a call based on the peer's MCR. If the callback fails by returning 0, the (blocking) accept call does not return; the connection is not established. Note that if *accept* is used without a callback, pending connections simply wait until *accept* is called[27]. The callback method can be used to signal a connection coming in asynchronously.

Accept returns the (authenticated) peer MCR in the *peer* argument, with all applicable ScID fields (*peer_id, zone_id, aos_endp.scid*) filled in.

For *connect*, the MCR should minimally contain the TCP/IP address and port of the (AOS or middleware) process being connected. Usually, the *zone_id* field is filled in, and if applicable the peer's AOS *index* field. Any missing applicable scid fields are filled in after authenticating the peer process. Any ScID that is specified before connecting, will be checked; if there is a mismatch, connect (or accept) will fail. A *close* call can be made to close a connection at any time.

[27] Perhaps a timeout should be added to connect so that a connecting middleware process's thread does not block indefinitely. This is not currently implemented. A connection may timeout automatically in cases when *accept* takes too long.

Send/recv write or read data from a connection, respectively. *Recv* takes a Boolean *block* argument; if *TRUE*, the call blocks until data is available. Since the underlying substrate is a stream, *recv* will return when *any* data is available, not necessarily the number of bytes specified by *len*. *Len* specifies the maximum number of bytes to return in *buf.

Select has slightly different interface and behaviour than BSD socket select, but does the same: poll the status of a set of descriptors. The difference is due to the semantics of SSL, in particular of the internal symmetric cryptography used: most symmetric key algorithms are block ciphers [36]. An SSL read operation will only return when sufficient decrypted data is available to decrypt a *block* of data (and verify its authenticity using its message authentication code), while a BSD socket select call returns when *any* data is available. The distinction is relevant because if data is available on the underlying socket, it may not yet be decryptable. The ZAC *select* call only returns when data is available in decrypted form.

A distinction with socket select is that ZAC select takes lists of descriptors (integers), while BSD socket select take bitmaps: descriptors in ZAC are not standard UNIX file descriptors within a number range up to 1024 (bitmaps in UNIX socket select are typically 1024 bits), but integers that can have any value. Other than that, the ZAC *select* call functions are the same as BSD *select*. It is used intensively, for example by the RPC layer to check if listen descriptors or connections are readable (*rd* set). A set of *descriptors* can be passed to *select* by reference. Any descriptors with *readable, writeable,* or *exception* status are returned, unsorted, in the beginning of the array; other descriptors are set to 0 by select. Like BSD socket select, a NULL argument specifies an empty select set.

The *block* argument ensures that *select* will only return if one of the descriptors in the *select set* is signalled with the relevant status. *Listen* descriptors can also be passed in a select set: an *rd* status on a listen descriptor indicates that there is an incoming or an accepted connection (the latter if an accept callback was set), and that *accept* can be called on this descriptor to obtain the connection descriptor; connections not accepted by the accept callback will not be visible in *select*. Placing both listen and connection descriptors in a (*rd*) select set, is an often-used way to implement a service using one or only a few threads.

5.1.5. Preventing AOS-level man-in-the-middle attacks

The basis for authentication in Mansion are self-certifying identifiers such as ZoneIDs, which are available at the application layer. The middleware can authenticate processes end-to-end at the application level using ScIDs.

AOS provides a means to delegate functionality for setting up communication channels from the middleware level to AOS. If the middleware processes at both sides trust their AOS kernel, AOS can provide confidentiality (over wide-area connections) for them. However, middleware ScIDs are not known to AOS. AOS takes care of encryption, and the ZAC layer takes care of authentication. This requires a challenge-response protocol on top of a (previously set up) AOS-level connection, to authenticate the peer process.

A naive implementation can introduce a vulnerability. When the underlying AOS kernels are not be authenticated as being the AOS kernels that the communicating middleware processes use, data may be routed through a man-in-the-middle and decrypted there, without the middleware or its local AOS noticing it.

The problem is sketched in Fig. 18. Middleware process 1 intends to set up a connection to middleware process 2, over AOS. Because the AOS ScID is not known in advance, the *aos_endp* information in the MCR is incorrect or can be modified by a man in the middle. AOS 1 may be connected to AOS 3, instead of to AOS 2.

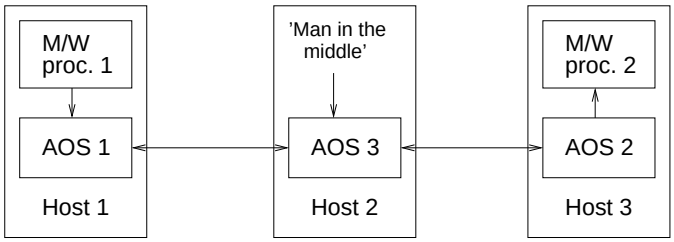

Fig. 18. A conceivable man in the middle attack on confidentiality when using a multiprocess communication stack

The solution is quite simple. If the top-level processes are mutually authenticated end-to-end using ScIDs, these processes can, as part of the authentication handshake, provide (authenticated, signed) information to their peer that makes it possible to verify that the underlying communication channel is not routed through a man in the middle. Practically speaking: a middleware should pass its own AOS kernel's ScID to its peer in an integrity-protected/authenticated way so its peer can compare this ScID to the ScID that its AOS kernel authenticated as its peer AOS ScID. After this check is done, both middleware processes can fully rely on their local AOS kernel to encrypt all traffic. If the locally obtained AOS ScID does not match the ScID obtained from the peer, ZAC aborts the connection with an error. Because AOS implements a secure channel abstraction, no further authentication checks are required as long as the underlying connection is not closed.

In ZAC, the above is implemented by starting an open SSL RSA-based challenge-response over the AOS connection; after this, ZAC sends an authenticated copy of its own MCR, including the AOS ScID, to its peer. As this information is signed using the peer middleware process' key, it can be trusted.

Note that the above mechanism may be relevant to authentication in multiprocess communication stacks in general. For example, trusted visualisation approaches sometimes make use of different layers (e.g., a virtual machine monitor, an operating system, application) which need all be trusted when communicating with an application [40]). Further treatment of this scenario is outside the scope of this dissertation. However, to our knowledge the use of ScID-based end-to-end authentication to authenticate multiprocess communication stacks is novel and has not been presented elsewhere.

5.2. The RPC layer

The RPC layer is used pervasively in Mansion. The middleware is essentially RPC-structured: communication between middleware processes, including object invocation, takes place using basic RPC invocations. Also, agents communicate with their MMW over RPC, that is, Mansion API stubs and object interface stubs use RPC to communicate with the MMW. Interagent communication also uses RPC internally (Sec. 8.2.13).

RPC implements communication between middleware processes. A process sends a request, and a reply indicates the status of the request (e.g., processed *OK*, *error*), and if applicable the result of the request (e.g., a contact record as the result of invoking a method on the location service). An advantage of using a single RPC layer compared to designing low-level protocols is that stubs and skeletons, to marshall request and replies, can be generated automatically by an RPC compiler. The RPC layer (particularly, the RPC compiler), avoids manual definition of wire formats for requests and replies.

Our RPC system is layered on top of the ZAC layer. Authenticated, secure connections are used to ship invocations over; both the RPC service and the sender of a request are authenticated.

RPC calls may be nested to implement layered functionality. For example, invocation of objects are RPC invocations on a layer in the Mansion Object Server (MOS), invoked using an RPC interface of the MOS accessible to remote clients (chapter 7).

5.2.1. Data representation

RPC is layered upon a standard data representation layer. A standard for data representation is required for invoking methods over the Internet, in particular when the system has to be portable across different architectures.

Mansion marshalls arguments eXternal Data Representation (XDR) [97], which is also used in SunRPC. Alternatives are available, for example the object marshalling format used in CORBA's IIOP system [86], or the format used for Java object serialisation in Java-RMI [68].

A primary reason to choose XDR is its minimality and simplicity. In particular, XDR supports only primitive data types, such as integers and byte arrays. It is important to avoid complex data types (for example as used in Java and IIOP), since the RPC system should be portable across languages and platforms. Minimality means that it is straightforward to implement stubs for different languages using a simple RPC interface definition language, using XDR for its data representation. Writing an XDR compiler is not very complex.

A disadvantage of XDR is that binary data such as integers are marshalled in network (big-endian) byte order, even when the machines on both sides of the wire represent integers in little endian. This could be alleviated by avoiding network byte order conversion if not required, but this would require both sides of a connection to exchange information about the

byte ordering they use, which increases complexity. Combined with the fact that most little endian machines (in particular, x86) have very efficient byte-swapping CPU instructions, the advantages of having a simple, uniform data representation system outweighs the (negligible) performance penalty.

5.2.2. Programming Mansion XDR

Mansion implements an XDR compiler that uses SunRPC-like struct definitions[28] in a file as input. In contrast to the standard SunRPC XDR compiler, the XDR compiler written for Mansion explicitly allows for stacking protocol headers by nesting XDR structs.

The XDR runtime library contains functions to create (allocate) memory, as well as conversion routines for standard data types such as 32 bits and 64 bits integers, booleans, and strings (which do not require byte conversion). The XDR compiler takes a struct definition as input to create a runtime library that can convert these structs to and from the XDR wire format at runtime.

In contrast to standard SunRPC XDR, variable-sized (opaque) byte-strings may also be defined in XDR definitions, where by convention a named field in the XDR struct defines the size of the string at runtime. Although minor syntactic differences exist between the way in which data is specified in Mansion RPC versus standard XDR, the wire format is fully compatible with SunRPC XDR.

An example of a simple Mansion XDR struct defining a variable-sized array is shown below. The XDR compiler translates this struct to a corresponding wire format.

```
string_str {
   int a;
   char[a] b;
}
```

A variable-sized character string *b* is defined; the runtime library generated by the XDR compiler ensures that *a* is pre-pended to the internal representation of array *b*. Client and server should agree on the XDR definition: the receiving party must ensure to allocate sufficient data to receive the character array. In the current implementation, the receiving library automatically allocates memory for *a* and *b;* a maximum amount of allocateable memory is defined in a global XDR header file. Sending and receiving party should agree on this maximum. If it is exceeded, a marshalling error is generated at the sending side.

As the result of XDR compilation, the Mansion XDR compiler currently creates C code that contains the data conversion routines. These are compiled and called by the RPC layer at runtime. Header files are created that contain the C structs and type definitions of arguments used by applications that use RPC. C++ stubs can also be generated. Note that because the

[28] See RFC 1057. http://www.ietf.org/rfc/rfc1057.txt

XDR representation is simple, it is straightforward to implement a compiler to create marshalling routines for another language, such as Java. An RPC compiler for Java is not currently implemented. However, XDR marshalling routines were implemented manually using an identical wire format when implementing a Java implementation of AOS (see chapter 4), demonstrating feasibility of the approach.

5.2.3. The RPC interface definition language

An RPC interface is defined and generated from an IDL file. The Mansion RPC IDL language is somewhat more flexible and intuitive than standard RPC in terms of syntax, however, the end result is the same. An example of an IDL file used to create an RPC interface—part of the Mansion API interface—is shown below:

```
scid_t {
   char[20] scid;
}

mcr_t {
   char[16] ipv6-addr;
   short port;
   scid_t aos_scid;
   int aos_index;
   scid_t peerid;
   scid_t zoneid;
   int index;
}

interface MansionAPI {
   ...
   int rmo_bind(out: mcr_t invoke_mcr);
   int object_bind(in: int eid, out: mcr_t mcr);
   int object_unbind(in: int eid);
   ...
}
```

The Mansion API is implemented as an RPC service which is instantiated by the main MMW process. This service is called by agents when they invoke methods on the Mansion API runtime library (see Sec. 8.2.8).

As depicted above, the IDL contains special markers for **in** and **out** parameters; for example, the *object_bind* call takes an *int eid* as an ‘‘in’’ argument (this indicates the *EntityID* of the object relative to the agent’s current room). The *object_bind* calls return an MCR as an ‘‘out’’ parameter, if the call is successful. Note that the above definition contains a definition of the MCR, *mcr_t* (Fig. 16) which contains a nested definition of the *scid_t* type. Note that the *aos_endp* struct is included inline for clarity; the resulting wire representation is the same using a nested struct definition.

A return value is also specified, typically an *int*. By convention in Mansion, negative return values are reserved for error codes, where a few specific (known) negative numbers are reserved for predefined RPC or ZAC error codes, such as errors in marshalling arguments, memory problems, or connection related errors. If a negative number other than an error code may be a result of a successful method invocation, an out parameter should be defined for this return value, instead of letting the call return it.

The caller of a method should define sufficient memory into which to copy *out* parameters. In the C/C++ stubs generated by the RPC compiler, out arguments are passed by reference. Clearly, the definition of methods and arguments should be such that no more information will be returned than can be copied into the user-provided buffer. The underlying RPC system will not read (from the ZAC connection) and extract more data to the buffer than specified in the interface.

The IDL compiler provides a construct to help handle variable-sized *out* buffers. The construct looks like this:

```
int read_data (in: int len, out: char[len] buf);
```

Len defines the length of the variable-sided array; the XDR unmarshalling routines generated from this example ensure that no more than *len* bytes are read into *buf*.

5.2.4. RPC service management

A Mansion RPC service is implemented using an RPC invocation infrastructure that uses skeletons generated by the IDL compiler. This section describes the RPC service management infrastructure.

An RPC service listens for incoming connections or invocations. This endpoint is created by the RPC service management interface using ZAC[29].

A summary of the most important RPC service management calls is given in Fig. 19. The basic calls used to register an RPC service or connect to it, are *rpc_svc_create* and *rpc_svc_connect*. The *rpc_svc_create* and *rpc_svc_connect* calls, respectively create a communication endpoint for invoking a service and for connecting to it. The calls instantiate the data structures required for handling XDR marshalling/unmarshalling for RPC request and replies, and the underlying communication system—associated with an RPC service (at both sides), is a thread that listens for incoming requests or replies and handles them. A call exists to register a thread pool with the communication subsystem, which allows handling of requests or replies using a separate thread (not shown). The *mcr* argument is used to connect to a service. The request/reply pointers are pointers to a struct containing the XDR *type definition.* This definition is used to instantiate the data structures which contain an XDR-marshalled request or reply.

[29] To our knowledge, no SunRPC implementations exist that are layered on top of SSL/TLS at the time of this writing.

Method(arguments)	Description
rpc_svc_create(substrate, port, *mcr, request, reply, *upcall)	Service creation + register upcall. Service's MCR is returned in mcr
rpc_svc_connect(*mcr, request, reply, *upcall)	Connect to service/MCR, register callback
rpc_accept_cb(*peer_mcr, *cb)	To accept or deny peer's connection
rpc_request_create(*rpc_svc)	Returns rpc_client_inv
rpc_request_send(*rpc_client_inv)	
rpc_reply_send(*rpc_client_inv)	
rpc_reply_await(*rpc_client_inv)	

Fig. 19. Overview of the RPC service management and invocation interface

A request/reply data structure is instantiated using the call *rpc_request_create*. This call uses the data structure returned by *rpc_svc_create/connect* as an argument, and returns a pointer to a data structure *rpc_client_inv*. The generated marshalling routines use this data structure to push or retrieve arguments for a given RPC invocation and to (un)marshall them.

To handle (incoming) requests, the RPC interface provides a number of mechanisms. A basic callback can be registered using which incoming ZAC connections can be verified. An MCR containing (ScID) information about the peer process is returned for each incoming connection; the callback function must indicate if the connection is accepted or not. A connection has to be accepted before RPC calls are allowed. Each service can have its own access control rules, that is, register its own callback.

The *rpc_svc_create/connect* calls allow a caller to register an *upcall*, which is a callback function. If an upcall is registered, it will be invoked whenever a request (or reply) comes in; this allows for asynchronous handling of incoming requests or replies. As an alternative to registering a connect upcall, a client can call *rpc_reply_await*, which simply blocks until a reply returns or an error occurs (e.g., as the result of a disconnected socket or an error at the RPC service side). This allows for synchronous RPC method invocations.

Although the above discussion does not give a very detailed overview of the implementation of the RPC layer, it shows how an RPC interface and appropriate XDR types are registered to allow for marshalling and unmarshalling requests and results, and to allow for registering the RPC service.

5.2.5. Connection-oriented RPC

Connections are not a natural aspect of RPC systems, as traditional RPC is stateless and often layered upon a connectionless UDP substrate. The connection-oriented nature of our RPC system is due to the underlying ZAC connections.

Like most RPC systems, our RPC system is in fact connectionless from the point of view of the RPC service: each call is handled independently. The RPC service can obtain a

peer's MCR independently to determine whether it allows the invocation and possibly to keep state per client, but it need not be aware that reliable, ordered *connections* are used underneath. Also, if a client closes its ZAC connection, this is not noticed by the RPC service. RPC services including the ones currently implemented in Mansion are normally stateless and only act on incoming requests, although they could maintain state for specific clients between requests if required.

The RPC system comes with an IDL compiler and XDR marshalling routines roughly comparable to SunRPC. Underneath the RPC layer, at the level of the ZAC layer, (1) a real connection-oriented communication channel is used, (2) because the accept callback is visible in the RPC interface, and (3) because the *rpc_connect* call contains XDR types which should match for client and server (even though this is currently not verified at runtime).

In all, usage of the term ''RPC connection'' is somewhat ambiguous. Our RPC system combines connection-oriented with connectionless properties.

5.2.6. Security of RPC

The XDR types of the *xdr_t* arguments to the *rpc_svc_create* and *rpc_connect* calls should match. If not, the server or client may decode (unmarshall) garbage. However, mismatches can occur, as checking of types cannot be done securely at runtime: a malicious, erroneously programmed, buggy, or wrongly connected client can always affect the server side. A solution could be to have clients and servers check at connection time whether the RPC interface they expect matches the one at the other end of the connection. This is expensive and does not help against buggy or malicious peers.

In our implementation, client as well as server ensure that consistency checks on the content of XDR data in invocations and replies are made. Internally, an XDR message always contains a size field preceding any (variable or fixed-sized) data structure on the wire, which allows the handling routines to allocate sufficient memory for decoding, and to ensure that no more data is read than is allocated. Also, this size field can be used by the recipient of the data structure to see if it matches the amount of data expected, and this allows the recipient to drop the connection if not (this gives a very evident message to the other side of the connection that something was wrong).

Servers (or clients) may block when they receive less data than they expect or when it arrives too slowly. With mismatching XDR types, a receiver may block to read data that never arrives. Server and client should be programmed to ensure that they do not block indefinitely in these cases. The current Mansion implementation does not protect against these kind of problems. The best measure that can be taken in the current implementation is that a service (or client) can register upcalls with a pool of handler threads, such that incoming data from several clients can be handled independently by different threads, avoiding that one blocking thread blocks all other handler threads. This also does not help against malicious or coordinated (Distributed) Denial of Service [(D)DoS] attacks. To ensure protection against

those our implementation could be re-engineered, for example to drop connections after a timeout. However, even then it appears improbable that one can defend the system against all kind of (D)DoS attacks.

.

Chapter 6
The Mansion Jailer

Agents may be malicious. Agents may try to steal information such as local files, gain root access, attack processes of other users, penetrate the (local) network to gain access to other systems, or mount a—distributed—denial of service attacks on other systems. Further, if no specific action is taken, agents can easily exhaust resources such as CPU time, memory or disk space. As an exercise, see what happens to your Linux system if you implement a C program with a recursive signal handler, that is, where the handler sends a signal to activate itself. Save your files and get ready to hit the reset button.

There are two main defenses against malicious agents in existing systems:

- **Code signing.** Agent code is signed, to attest that the code can be trusted.
- **Language-based sandboxing.** Agents are executed in a language-based **sandbox** environment (interpreter), possibly specifically modified for mobile agent execution (e.g., [59, 55,87]).

Neither of these two approaches provides sufficient protection. Below we explain why, before describing the jailing system that was designed for Mansion.

Code signing. Although code signing has been used in some agent systems that do not use mobile agents (e.g., JADE [32], the approach is mainly used in desktop grids such as XtremWEB [34]. Here, the focus is large-scale computation where, often, only a limited set of programs are admitted that each need a large set of resources. Code signing is also used in mobile agent systems, but here code signing is generally used only to bind agents to an identity in a secure way, in the form of an *Agent Passport* [91]; this approach is also used in Mansion.

As a mechanism to provide security against malicious code, a code signing approach does not scale. Even if some agent's code could be verified and attested as ''secure,'' for

example by its author or an independent code certification authority, many binary agents will link with third-party libraries or with code written by other authors than the specific piece of code which implements the agent's task. What if the compiler has a bug in it and produces flawed object code? Who will want to verify the code for many users, in particular for arbitrary code, and verify that it does not contain any vulnerabilities? Even so, can we be sure that (dynamically linked) code cannot be exploited? With estimates from 1 to 50 bugs per 1000 lines of code, it seems impractical if not theoretically impossible to ensure that any given agent will contain no potentially exploitable or harmful pieces of code. There exist some automated code verification techniques, but these often do not work for legacy programs, or are complex or inefficient [74]. Thus, code auditing/signing will not suffice for a system that aims to support heterogenous, customiseabe agents for thousands or maybe millions of users.

Language based sandboxing. An advantage of language-based sandboxes, effectively interpreters, is that they can be made to support strong mobility. Another advantage is that they often provide an operating-system and architecture independent execution environment. They are also useful for experimenting, since it is relatively straightforward to modify the interpreter to extend or alter the language.

Language-based sandboxing are the dominant execution environment for mobile agent systems and have been explored extensively in this context. Many agent systems use *agent servers* (Sec. 8.2.5) that implement a language-specific sandbox. Due to its platform-independent nature, Java is common, but other languages have also been used. These include Scheme, Python, TCL and Telescript.

A disadvantage of language-based sandboxes is that it is difficult to get their security model right. The JVM—one of the most extensively used systems for secure, platform-independent code execution—still suffers from security issues today, which are mainly caused by shortcomings of the Java language [14]. Further, as soon as some interpreter is adapted to embed agent system-specific instructions or protections, a large burden is placed on the agent system programmer, to adapt the interpreters—if applicable, for multiple interpreters—to other operating systems or language versions. From a software engineering and a security perspective, it scales badly: due to the divergence of supported features and versions, it is likely that bugs creep in. In other words, it becomes hard to maintain a *trusted compute base* consisting of language-specific sandboxes.

An additional problem is that policy management languages for configuring sandboxes differ between languages and interpreters, and are often complex [43]. And finally, language-based sandboxing does not support binary compiled programs—which still form the most efficient type of programs today.

The above argues for a low-level, simple protection system, close to the operating system for efficiency and security, with a straightforward and language-independent policy model that supports binary code as well as interpreted code. This is described below.

6.1. Introduction to the jailer

Mansion supports agents written in arbitrary languages. This includes binary (legacy) applications. Current-day operating systems are not well equipped to protect a system from malicious processes (in particular, these execute under a user's ID and can thus access or damage files of this user). Therefore, a mechanism is needed to ensure that agents cannot harm the local system. This mechanism is called **jailing**.

This chapter describes the design, implementation, and performance of the Mansion jailer[30]. An important contribution of the jailer is that it runs completely in user mode on unmodified UNIX (Linux) kernels, while being secure against race conditions that rendered earlier user-mode jailing systems insecure [39]. It allows for efficient execution of arbitrary programs. The jailing model allows system calls given that these do not violate a user-defined policy, and do not, for example, try to send signals to arbitrary programs on the machine, or read arbitrary files which it has nothing to do with. This simple but powerful *jailing model*, combined with a customiseable policy, determine what agents in a jail are allowed to do. Most programs work in a jail with an unmodified default policy.

This chapter explains the jailing model, implementation, and performance in detail. The jailer is used to jail agents, and possibly objects if their implementation is not trusted. The jailer is a stand-alone program used by Mansion[31]. It has been designed for portability. It runs not only on standard Linux kernels using the *ptrace* debugging interface (for intercepting system calls), but also on a modified Linux kernel that uses a custom-built tracing system. Because it runs on two tracing layers with otherwise identical machines and operating systems, it becomes possible to compare the performance of a jailer with an optimised tracing layer, compared to one that used the *ptrace* system available on every UNIX system. The resulting measurements are described in this chapter.

6.1.1. Approach

Operating systems currently do not provide sufficiently fine-grained protection mechanisms to protect a user against the programs he or she executes. The UNIX protection model is based on a discretionary access-control model, where all programs executed by a user inherit the user's permissions with regard to accessing resources, such as files. When using UNIX as a platform to execute untrusted programs, it is important to be able to automatically protect a system and its users against such programs in a fine-grained way without requiring user intervention or complex configuration procedures.

[30] This chapter is based on an article which appeared in 2007 [82].

[31] The jailer has recently been re-implemented to simplify its code base, making use of Linux-specific features to allow for certain optimisations and leaving out certain complex solutions for problems that occur only rarely [128]. The jailer is a stand-alone program so it can also be used in other systems, such as desktop grids, or as a command line tool for confining untrusted applications downloaded from the Internet.

System-call interception based jailing systems are most often based on a kernel-level tracing mechanism (e.g., *ptrace* or */proc)* that allows a trusted jailer to intercept all system calls of its child process(es), and accept, deny, or modify arguments of the system calls made by a jailed process before the kernel proceeds with executing the system call. A number of jailer designs have been described since the first jailing system was described in [117].

The most common use of jailing systems is to protect systems against untrusted (downloaded) executable or interpreted code [117, 96, 41], and for intrusion detection [49]. All jailing systems come with a policy that describes which parts of the local file system may be accessed, and which network addresses are reachable by the jailed processes.

A number of jailing systems require modifications to the operating system to function securely. Jailing systems that are implemented in user-mode, using the *ptrace* or */proc* debugging facilities offered on standard UNIX, also exist. However, existing systems suffer from several race conditions, which allow an attacker to bypass the jailer's control mechanisms [39]. Over time, jailing systems have become more mature, and most in-kernel systems can be considered secure with regarding to enforcing the specified policy. However, some implementation issues remain for jailing systems which are implemented in user-mode, by using, for example, the *ptrace* or */proc* debugging facilities offered by the operating system.

This chapter describes novel solutions to these race condition problems. These solutions allows complex programs, including multithreaded programs that make use of IPC mechanisms and signals, to be jailed effectively. The jailing system presented in this chapter provides sufficient control to allow for effective confinement of untrusted programs using standard system call tracing mechanisms available on most UNIX systems.

This chapter is organised as follows. First, it describes the design goals of our jailer, and discuss a number of practical issues that make system call interception systems difficult to implement securely. Next, it discusses the jailing model and policy in some detail, and how information leakage from a jail to the outside world may be prevented. It then discusses how the system is implemented and aspects regarding portability. Finally, this chapter describes performance measurements and related work, and it draws some conclusions.

6.1.2. Design goals

The design of our jailing system is motivated by the need for a secure confinement system for mobile agents in the Mansion system. Agents are viewed as code (e.g., a precompiled binary) that can migrate to different machines if required; agents are just executable code. The jailing system should be realised as a user-mode program that can run on standard UNIXes, initially Linux, which is simple to configure and use, and which allows us to guarantee that an untrusted agent cannot attack the system or read files from the system to which it does not need access. In general, agents in different rooms (particularly, confined rooms), should not be able to communicate information directly to each other. By default, an agent in one jail should not be able to communicate with an agent in another jail. Although the current

implementation does not prevent all types of information leakage (in particular, covert channels [11]), jailing may provide a basis to prevent information leakage in practical scenarios, and thus provide a basis to control information flow.

Although the jailing system can be (and is) used in Mansion, it is in fact a stand-alone program that can be used to confine any untrusted application. The main design requirements for our jailing system—note that these are separate from Mansion as a whole, as the jailer must be usable as a stand-alone system—are:

- The system should be simple to use. In particular, it should be straightforward to specify a sufficiently secure default policy to confine untrusted applications, and preferably it should be possible to run applications in a jail without modifications.
- The system should be portable to different UNIX systems, and runnable by any user without requiring root privileges. As jailer is intended to be usable in a distributed system, it is not restricted to a single UNIX version (for now, Linux version). System administrator intervention, for example, to patch or reconfigure the operating system before a regular user can use the jailer, also needs to be avoided.
- The system should support all types of applications. These can range from shell scripts spawning many processes that communicate via pipes or other forms of IPC, to interpreted programs, to heavily multithreaded programs such as a JVM, and more. These programs should run out of the box.
- The system should be secure in view of race conditions or coordinated attacks by malicious multithreaded programs.
- It should be possible to confine processes, including multiple programs executed by the same user. Direct communication channels between processes in different jails should be prevented, unless explicitly allowed. (Covert channels are not considered).

The first requirement implies that specifying a jailing policy should be a straightforward task. A simple, high-level policy language is defined, which can be used to adapt the policy to in particular, the local file system's directory structure (Sec. 6.2.2). The second requirement implies that the system should be runneable using standard operating system primitives, such as *ptrace*. The confinement property requires the jailer to keep track of a number of system calls and their arguments to prevent jailed programs from directly communicating with other jailed programs. Programs or threads within a jail should be capable of using most UNIX calls for interacting with the file system and for inter-process communication, however. Only this way will complex, modern programs be able do useful work in a jail (Sec. 6.2.1).

Another important issue for jailing systems is usability. This impacts both the way in which policies are defined by a user, and the jailing system's implementation. The system should be usable by regular UNIX users without requiring system administrator intervention or operating system or kernel modifications. The system described in this chapter has been

designed such that it can be implemented using rudimentary debugging support available on every major UNIX system, such as the *ptrace* or System-V's */proc* system call tracing interfaces (Sec. 6.2.3). Although earlier jailing systems are built using these tracing interfaces [8, 117,49], none of these were fully secure. The system presented in this chapter is the first which is completely secure even in view of race conditions that rendered earlier attempts to build jailing systems in user space insecure [39].

6.1.3. Overview of terminology and technology

The terminology used in this chapter is the following:

- The **jailer** is a trusted process that monitors an untrusted application and enforces a policy on the user's behalf.
- A **prisoner** is an untrusted application that is monitored by a jailer and is forced to adhere to a predefined jailing policy.
- The **tracer** is the interface offered by the operating system for debugging / tracing an application. Every major UNIX system to date provides one or more tracing interfaces, such as *ptrace* or System-V's */proc* interface.

The basic idea of system call interception is demonstrated in Fig. 20. Most if not all current UNIX systems provide some form of debugging support that allows for catching and inspecting the system calls that an application makes, allowing for inspection of the system call's arguments. The primary example, which is rudimentary but still often used (e.g., by gdb), is the *ptrace* system call.

This section assumes *ptrace* as the underlying system call tracing interface, as it is the most primitive tracing system currently available and demonstrates the minimal requirements of a user-level jailing system.

Ptrace is a system call that allows a parent to monitor its child's behaviour. When a traced process makes a system call, this call is automatically trapped by the operating system and reflected to the parent process, which can then inspect its child's current register set and system call arguments. Based on this, the parent (jailer) can decide whether to let the system call proceed or whether it should return an error without being executed. *Ptrace* allows the jailer to change the value of registers that contain the system call's arguments, before letting the kernel execute the call. *Ptrace* intercepts every system call just before it is executed by the kernel, and right after executing it. Only after the jailer agrees to let the process continue on both the pre and post system call event, is the the prisoner thread resumed.

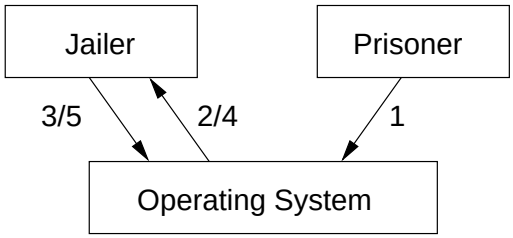

Fig. 20. General positioning of a system call interception system showing a jailer and a traced prisoner. When a prisoner makes a system call (step 1), the operating system suspends the invoking thread and reflects the system call to the jailer (step 2). The jailer inspects the system call's arguments (e.g., by inspecting register values or values on the prisoner's stack) and decides if it should allow the system call. It informs the operating system of its decision (step 3), which results in the system call being continued or an error being returned to the prisoner. Step 4 and 5 repeat step 2 and 3 after the system call has been made, so that the jailer can inspect the result of the system call before returning control to the prisoner.

6.2. Threats and vulnerabilities

Fig. 20 illustrates a significant problem in all system call interception based jailing systems that support multithreaded applications or processes that use shared memory. When the operating system suspends the invoking thread and reflects the system call to the jailer (step 2), the jailer has to make a decision on whether to allow the system call based on its arguments. These arguments often contain a pointer to a string (e.g., a file name) in the prisoner's address space, which must be de-referenced and checked by the jailer.

Between the time that an invocation is made (step 1/2) and the decision has been passed back to the operating system (step 3), a different thread of the prisoner (or a process which has access to the prisoner's address space) could have modified the argument in the original thread's address space. In this case, the system call would end up using the modified system call argument rather than the argument checked by the jailer. This race condition is called a **time of check to time of use (TOCTOU)** race, and it is a realistic threat for all jailing systems that are intended to support multithreaded programs or programs that use shared memory. This threat applies to system calls that take a file name as an argument, but also to a connect call that takes an IP address as an argument, for example.

Several solutions for the shared memory TOCTOU race have been proposed for existing system call interception based jailing systems [41,117,96]. The most secure approach among current systems is to let the kernel create a safe, normalised, copy of the arguments before reflecting that to a user-level policy enforcement module [96], to make sure that the kernel will use the same arguments as those passed to the policy module when it executes the call. Another approach is using a delegation approach, where the jailer invokes the system call on behalf of the prisoner [41]; for the latter approach, some portability issues were reported [117]. Ostia, the research system that presented the delegation approach first,

requires installation (as root) of a kernel module to implement part of its functionality.

TOCTOU race conditions are a challenge to solve for user mode systems that rely on existing tracing mechanisms such as *ptrace* or */proc*, which provide no protection of the arguments of a system call at the time of inspection. The approach closest to solving the shared memory race outlined above is described in [49]. This solution is based on relocating a system call's argument to a random location on the caller's stack before checking it, so that another thread in the child's address space is unlikely to be able to find and replace this argument. However, it is certainly not impossible for another thread to find such a relocated argument and replace it[32]. Other jailing systems simply disallow thread creation, or suspend all threads of a jailed process while a system call is being evaluated [3]. Both approaches significantly limit the applicability of such systems for executing modern thread-based applications.

Certain file system race conditions have also been documented for system call interception systems [39] and for operating systems in general [64]. These race conditions are again caused by a lack of atomicity between the argument checking and system call invocation steps. Between the time that a system call's file name argument is verified by the jailer and the time that the system call is executed in the kernel, another prisoner thread (or even an altogether different process) may have substituted a part of the underlying file system path for a symbolic link to a directory outside the paths that are allowed by the policy, without the jailer being able to detect this. For example, the prisoner may invoke open with /tmp/user/temp/passwd in the allowed path /tmp/user/temp, and substitute the temp component for a symbolic link to /etc, hoping that the system call will be executed just after that, resulting in /etc/passwd to be opened. Changes to the current working directory while a system call is being evaluated by the jailer can evoke similar race conditions. An excellent overview of these and other vulnerabilities in system call interception based systems is provided in [39].

A weakness of most existing jailing systems is that the only way they can handle system calls is to either always allow or deny them altogether, or to conditionally allow or deny the system call by comparing its argument with, for example, a set of file names or network addresses in a user-provided policy file. When the policy provides insufficient information, the jailer's only choice is to allow or deny the system call always, irrespective of its arguments. Some systems make a callback to the user so he or she can make a decision to allow or deny the system call based on information provided by the jailing system [96]. Calling the user is not only inconvenient, it is also infeasible when a large number of jailed processes are running simultaneously on a system. In addition, a user does not often understand the meaning of the arguments of every system call of the UNIX API, and the potential side effects of allowing the call. In particular, system calls such as IPC calls or the kill system call take integer arguments which are determined at runtime. Often, these arguments indicate some kernel

[32] Winning this race is not as far-fetched as it may seem, especially since the prisoner knows the argument which it originally specified itself. By creating a large number of threads, a prisoner can increase the probability that one or more of its own threads are scheduled between the time that the jailer relocated the argument and the time that the corresponding system call is executed by the operating system, so that it gains more time to search for the relocated argument and replace it.

object of which a user cannot determine whether access to it should be allowed or not. Although some kernel-level security architectures (e.g., SElinux [4] and DTE [122]) allow for automatic tracking of system call arguments this way, this is does not hold for the majority of existing system-call interception jailing systems. The jailing system should provide a model for making policy decisions based on such arguments automatically, to make sure that a prisoner cannot escape its confinement.

6.2.1. The jailing model

To address the requirements and threats outlined above, our system provides a clear jailing model, which distinguishes an application's allowed actions *inside* a jail from actions that influence the world *outside* the jail. Within a jail, the jailer allows the full UNIX API, including IPC mechanisms such as shared memory, to be used by all processes within the same jail, with the exception of root privileged calls. To make sure that a prisoner cannot export information to the outside world, the jailer keeps track of which communication endpoints or IPC channels are created inside the jail, and only allows access to internal endpoints. The jailer also controls who may connect to a communication endpoint created in a jail, to prevent external processes to initiate a connection to a jailed process or IPC channels (see also Sec. 6.2.4). A prisoner cannot set up socket connections to arbitrary processes outside the jail. Jailed programs can send signals, but only to processes running in the same jail.

In addition to the built-in jailing model, each jail has a simple policy using which the user can define which file system parts a jailed program may access (read-only) by default. Every jail provides its prisoners with a private directory structure in which they can read and write files. This directory is protected from other users using standard UNIX protection mechanisms. A jailed process is by default started up in a (normally empty) scratch directory. Different jails are not allowed to have read/write access to directories shared with other (jailed) processes. This is an important constraint, as it prevents uncontrolled information flow between untrusted processes in different jails. In their own jailing directory, prisoners can write / read files and create subdirectories.

An important assumption of the jailer is that a user will jail all untrusted programs, to make sure that untrusted programs cannot access any files created by a prisoner in another jail. The user who has started up a jail is able to access the files and processes within this jail directly. This user can also send signals to processes in the jail, for example to kill a prisoner.

A jail always starts with a single program, but this program may (modulo policy) *fork* or *execve* other programs or create new threads. Child processes are executed in the same jail under the same policy as their parent. The jailing process hierarchy is shown in Fig. 21.

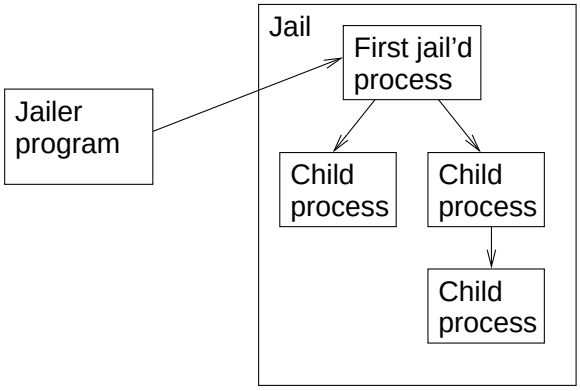

Fig. 21. A jail's process hierarchy. The jailer starts the first jailed process in its own jail, and controls this jail by enforcing the jailing model and the jail's policy. The first jailed process can create child processes (using fork and execve). These processes are now in the same jail as their parent and execute under the same policy as their parent. All processes within a jail can communicate with each other using UNIX IPC primitives (e.g., using pipes or shared memory primitives like *shmem* or *mmap*) or signals, or by writing files in their jailing directory. Communication with processes outside the jail is controlled by the jailer's policy.

The jail concept describes whether a set of processes may communicate with each other or not: processes within a single jail may freely communicate with each other, but communication with the outside world is not allowed unless explicitly permitted by policy. The jailing model enforces information flow policies even when the jailed programs run under the same UNIX UID. Within its jail, the jailer allows almost the full UNIX API, including IPC mechanisms such as shared memory (but no root privileged calls) to be used by all processes. Because the jailer allows most UNIX calls to be used freely within a jail, it allows for execution of the majority of programs, even modern multithreaded and multiprocess applications, within the confinement rules of the jail.

6.2.2. The jailing policy

For modern (UNIX) systems with a large number of system calls, understanding the potential side-effects of any particular system call that a program can make is a hard (if not impossible) task even for well-informed users. When a user executes many untrusted programs, it becomes infeasible to generate a policy for each program individually. The jailing model simplifies the procedure for specifying a secure policy by providing default rules which are aimed at providing strict confinement of the jailed programs. The system has a built-in policy which allows harmless calls (e.g., *getpid*) by default, such that the user only has to consider high-level policy aspects for sensitive system calls, such as which files or directories a jailed program may access.

Each jail is associated with a single policy, specified at jail startup as a user-editable policy file. Different jails may use a different **jailer policy**, although in most cases a single policy is used for jailing all applications. A policy file's syntax resembles that of environment variables in a shell script. An example policy is shown below. This is an excerpt of the policy file used for the benchmark tests in section 6.5.

```
# -- Environment adaptation
ENVIRON=HOME=$JAILDIR
# -- Read-only  / read-writeable paths.
ROPATH=/usr:/lib:/bin
# Jaildir is read-write accessible.
RWPATH=$JAILDIR:/tmp/$USER/$JAILDIR/tmp/
# -- CHRDIR directives (see text) --
CHRDIR=/:$JAILDIR
# The '/' CHRDIR is escaped for /usr, /lib, and /bin
CHRDIR=/usr:/usr
CHRDIR=/lib:/lib
CHRDIR=/bin:/bin
# /tmp must be CHRDIR'ed (see text)
CHRDIR=/tmp:/tmp/$USER/$JAILDIR/tmp/
# -- Communication / IPC configuration
# Here we deny accept calls, but we can also allow (IP addresses) or use a
# command-line IPC escape
INET_STREAM_ACCEPT=DENY
# Allow TCP connect to a local web server (port 53) and to $IPCESCAPE,
# which contains addresses specified on the jailer's commandline.
INET_STREAM_CONNECT=127.0.0.1/80,$IPCESCAPE
# Allow UDP sendmsg to a local nameserver (port 53) and to commandline
# addresses. Note that in a confinement setting, traffic to a remote
# nameserver could conceivably be used to leak information to the outside
# world.
INET_DGRAM_SENDTO=127.0.0.1/53,$IPCESCAPE
# UDP recvfrom seems safe, as there is no way to export information via
# receiving a datagram. However, a prisoner could snoop on
# NFS traffic, which could be used as a covert channel by other prisoners
# if ROPATH contains a mounted NFS directory.
INET_DGRAM_RECVFROM=DENY
# Shown for completeness: Pipes, socketpairs, MMAP and other IPC calls are
# controlled by the jailer. Therefore it is safe to allow these system
# calls (they are allowed by default).
PIPE=ALLOW       # or DENY
SOCKETPAIR=ALLOW # or DENY
MMAP=ALLOW       # or DENY
# ...
```

Fig. 22. A (default) policy for the jailing program

A user only edits—typically once—those parts of the policy file that concern access to the local file system, such that jailed programs can access important files such as shared libraries. Every policy contains an explicit list of read-only and read-write accessible directories, called ROPATH and RWPATH. Directories that are not in either of these lists are not accessible to prisoners. As an example, a prisoner will normally be allowed read-only access to certain other directories (e.g., /lib or /usr/bin), so that it can load standard libraries or execute regular programs (e.g., a Perl interpreter or a JVM executed by a script). Note that

because these programs are executed by a jailed program, they are run in the same jail as their parent and are subject to the same policy.

In addition to the policy file, the jailer program accepts a number of command-line parameters that allow the user to specify, for example, a set of network addresses or ports that a particular jailed program may access. This makes it possible to change some parameters for the jail at runtime without changing the policy file, for example if the program needs access to a local DNS resolver or a local Web server. The policy has a few reserved variables which are set by the jailer program, to keep the policy file independent of a particular jail's settings. Examples are $JAILDIR and $IPCESCAPE. $JAILDIR contains the pathname of the jail's private jailing directory. to define one or more (external) TCP, UDP, or UNIX domain endpoints which may be connected to by the prisoner.

In addition to ROPATH and RWPATH, the policy contains a directive CHRDIR that allows the user to *change-root* the program's visible directory structure using a simple path-prefix substitution algorithm built into the jailer program. In the example policy, the prisoner has read-only access to /usr, /bin, and /lib, and read-write access to $JAILDIR and /tmp/$USER/$JAILDIR/tmp. A CHRDIR directive applies a change-root to the jailing directory (CHRDIR=/:$JAILDIR), so that when the prisoner attempts to access the root directory ''/'', it really accesses its private jailing directory. A CHRDIR directive is also specified for /usr, /bin/ and /lib, to escape the ''/'' CHRDIR directive, such that when the prisoner accesses a file or directory relative to these directories, it accesses the real /usr, /bin and /lib directories. The ROPATH and RWPATH directives are applied before applying the CHRDIR directive: ROPATH and RWPATH define the absolute, real paths that are accessible; the CHRDIR path prefix substitution is only provided to customize the prisoner's environment. This avoids that regularly used files or libraries have to be copied into the jail's change-rooted directory structure before the jail can be used.

Many programs (e.g., the Java virtual machine) require write access to /tmp. However, allowing every jail to use the shared /tmp directory for writing / reading files, would allow jailed processes in different jails to communicate easily through files in this directory. Thus, /tmp is change-rooted to a private directory /home/$USER/$JAILID/tmp, such that different jails use different underlying directories when they access /tmp.

All unknown (or unspecified) system calls are denied by default by the jailer. This also applies to some known system calls, for example because we were unsure about the potential side effects of a system call (for example, we currently deny a number of *fcntl* and *ioctl* subcalls). Also, system calls unknown at jailer compile time (e.g., system calls added to a new version of the operating system) are denied by default.

The policy file can be used to override default policy decisions hardwired in the jailer, or to allow unknown system calls using their raw system call ID. A user can find out about denied system calls by running the jailer in verbose mode. In our experience it is not necessary to override the default policy, as most applications and their libraries simply run as expected—even if some system calls are denied. All tested programs and their libraries are quite resilient to system calls denied by the default policy. This also applies to files not found

by the prisoner, for example files in /etc. All prisoners make at least a few requests (generally, through *libc*) for inaccessible files, but turn out to work fine when access to these files is denied[33]. We found that we could successfully run many different programs in the jail, ranging from simple UNIX commands like *ls* to more complex programs like the bash shell or Java, using the default policy shown in Fig. 22.

6.2.3. Winning the shared memory race

The most important implementation issue to solve is how to secure the arguments of a system call in view of the shared memory TOCTOU race conditions outlined in Sec. 6.2. This section describes the Linux solution first, and then explains how this solution can be applied to other operating systems. After this explanation, it describes the architectural design of the jailing system.

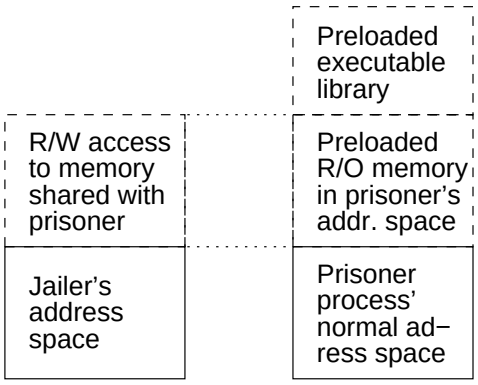

Fig. 23. The Mansion Jailer's Shared Read-Only (ShRO) memory solution for avoiding user-level multithreading and shared memory race conditions. The shared memory region is mapped in transparently (using *mmap* or *shmem*) by a library loaded at the time of prisoner startup using a standard ELF preload technique. In addition to the shared read-only memory (shRO), a small library with executable code for managing certain post system-call issues is also loaded in at prisoner startup time. The prisoner program itself is not modified, it is normally not even aware of its preloaded ShRO memory region or the library in its address space.

Fig. 23 depicts the solution. When a new prisoner process is executed, it is provided with a region of shared memory which is shared between the prisoner and the jailer. The prisoner has only read-only access to this region; the jailer can write into it. This shared memory region is called **shared read-only memory,** or **ShRO** in short. The shared memory is set up

[33] In practice, most refused system calls for the programs we tested were made by glibc. Glibc tries to open a number of files (e.g., in /etc) and tests a number of system calls, probably to test its environment so it can configure itself (glibc should be able to run on various versions of Linux); most refused system calls are made when the program is started up and when glibc is initialised. Glibc is quite resilient to refused system calls or unavailable paths on the system, and most tested programs work fine even under a very restrictive jailing policy, for example, only allowing (read-only) access to /bin and /lib. Some programs require access to specific files in /etc or /proc (Linux' process information database).

by using a library preloading technique (see section 4.3). During preload, the library contacts the jailer to obtain information about the shared memory region, after which it uses *mmap* to map this region in its own address space. Preloading avoids that we have to patch the prisoner binary. The preload library also contains some additional code which is required for handling certain post-system call processing tasks. These tasks are described in Sec. 6.4.1.

Once a system call is made, the system call is reflected as explained in Sec. 6.1.3. After this, the jailer fetches the arguments from the kernel using a standard (*ptrace*) call. An argument can be a file name or an IP address, for example, depending on the system call. The jailer makes a safe copy of the arguments pointed to in ShRO and adjusts the argument values (i.e., registers pointing to the arguments) in the Linux kernel to point to these copied arguments, my making a *ptrace* system call. Then the jailer does a policy check using the safe copy of the arguments in ShRO, and informs the kernel of its decision to let the system call proceed or not. If the system call is to proceed, the kernel now uses the updated system call registers, that is, pointing to the safe copy of the arguments, to execute the system call. No prisoner thread can touch the registers or the safe copy of the arguments.

The ShRO solution alone is not sufficient to prevent all TOCTOU race conditions. For example, there is a window of opportunity for a malicious program to substitute a file in its RWPATH (which has its canonical file name stored safely in ShRO) for a symlink to a file outside the prisoner's RO/RWPATH, between the time where the file name is stored in ShRO, and the time that the system call is executed. This race is prevented by serialising all system calls that can modify an object in RWPATH (i.e., all create/write accesses to elements in RWPATH) while an *open* system call is in progress. Serialising is implemented by placing a readers/writers lock around the open and modify/create/write-type system calls. This provides an effective solution to the shared file system TOCTOU races outlined in Sec. 6.2, making it impossible to mount symlink substitution or *chdir* based attacks from within a jail. (In theory, an attacker outside a jail could conspire with a prisoner to mount a symlink substitution attack; however, the jailing model ensures that different jails cannot access other prisoner's RWPATHs, and it assumes that a user jails all untrusted processes, excluding this attack possibility).

In Linux, the system call arguments are stored in registers at the time of invocation, and immediately copied to the operating system's process table when the system call is made. These register copies are secure from modification as they cannot be altered by the invoking process. However, the way in which system calls are passed to the kernel is operating system specific. Other UNIX operating systems (e.g., BSD) place the system call's arguments on the stack and pass the invoking thread's program counter (i.e., return address) and the stack pointer to the operating system, which fetches the the system call number and arguments from the stack. To secure system call arguments when these are on the stack, the jailer must copy the stack frame containing the system call arguments to ShRO memory, and modify the stack pointer in the kernel such that it will use the (possibly modified) stack frame in ShRO to execute the system call.

6.2.4. Preventing information leakage

Many UNIX system calls can be (mis)used to export information from a program in one jail to a program in another jail. For example, prisoners can escape their confinement by agreeing in advance on certain 32-bit *IPC tokens* prior to being started up in a jail. IPC tokens are used for identifying shared memory (*shmem*) segments, message queues, mutexes and semaphores in the kernel, and are specified by the program at creation time. IPC tokens have user permissions to ward off other users on the same machine, but processes executed by the same user (in different jails) can bind to a shared IPC channel using the IPC token. When the jailing system does not take special measures, a prisoner in one jail can bind to an IPC channel created in another jail or even by another user using a pre-agreed token. Using this IPC channel, a process in one jail can pass information to an agent in another jail, thus potentially bypassing information flow control rules imposed by the application. Existing jailing systems cannot automatically constrain such communication, except by always denying vulnerable IPC calls or signals. As many programs make use of IPC calls and/or signals, denying those calls makes the jailing system unusable for executing many programs. For example, a Java virtual machine makes heavy use of signals internally, so it will not run in a jail which denies access to the kill system call used for sending signals.

Using the jailer avoids having to configure a different security policy for each language run-time system or version thereof, even if that system would provide sufficient hooks for preventing information flow. This particular issue is solved by keeping track of the *pids* and IPC tokens used in one jail, and ensuring that prisoners cannot bind to an IPC channel that was not created in this jail or send signals to processes outside the jail.

There is also a vulnerability in the *stat* system call. Stat shows access times of files, even when these are only read. This means that when a prisoner in one jail reads from a file in ROPATH, a prisoner in another jail can stat this file's access time, leading to a covert channel. This issue is solved by zeroing the *st_atime* field[34] of the *stat struct* returned by *stat* using a post-system call intercept construction, which is discussed in Sec. 6.4.1. This is a costly operation, but fortunately this is only required when accessing files outside the jailing directory or RWPATH, as within a jail processes are free to exchange information, also by using file access times.

6.3. Jailer architecture

The architectural design of our jailer separates generic functionality (i.e., policy enforcement) from platform-specific functionality. The jailer is split up in two layers. The lowest layer is

[34] As far as we are aware, the *stat* system call has no other covert channels. The file size reported in the stat struct, for example, can only be modified by prisoners that have write access to a file. Since jailing policies are designed to avoid that prisoners in different jails have write access to shared directories or files, modification of file size by a prisoner is not possible. The distinct problem with the *st_atime* field which leads to it becoming a covert channel, is that it can be modified by *reading* the file. Different jails *do* have read access to shared directories, like /bin.

called the **interception layer.** This layer interfaces with the underlying system call tracing interface, e.g., *ptrace* or */proc*. Two interception layers have currently been implemented, one that uses *ptrace* and one that uses a specially-built in-kernel system call interception interface, called **kernel jailer.** Above the interception layer lies the **policy layer,** which takes care of enforcing the jailing policy. Both layers have access to a **shared memory manager** module, that manages the ShRO memory region. It has a high-level interface that allows the interception layer and the policy layer to allocate memory for writing system call arguments or stack frames into when required. The jailer architecture is shown in Fig. 24.

The interception layer handles the tracer-specific mechanism for attaching to a prisoner process, and it sets up the shared memory between the prisoner and the jailer. Prisoner code never runs uncontrolled. Before prisoner code is exec'ed, the forked child (which still runs the trusted jailer code) attaches to the jailer. From that moment on, it runs in a mini jailing environment. Then, the child exec's the prisoner binary. The prisoner is provided with a preloaded library when it is started up. This library is invoked by the ELF dynamic loader (ld.so) before the prisoner's code is run. This library sets up the ShRO region, connects to the jailer using a UNIX domain socket, and contains some code required for post-system call processing tasks as explained in Sec. 6.4.1.

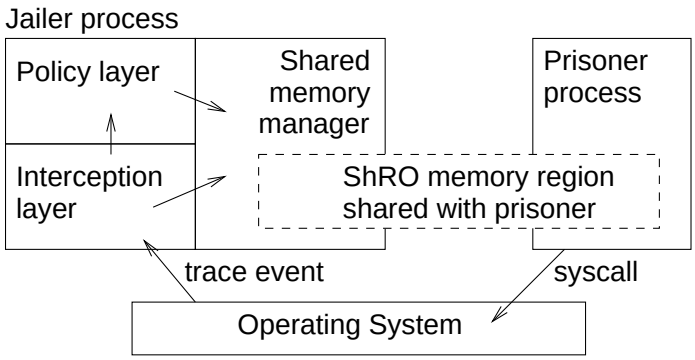

Fig. 24. The jailer's internal architecture. The jailer consists of an operating-system specific interception layer and a portable policy layer. A portable shared memory manager module manages the ShRO memory region(s) of the prisoner process(es) in the jail, and is used by both the policy and the interception layer. It is used by the interception layer to write a normalised copy of the system call arguments to, and by the policy layer to read these race condition-safe arguments from at policy evaluation time, and to write any modified arguments into if required. The interception layer drives the jailer by handling trace events from the operating system and by calling the memory manager and policy layer accordingly. The interception layer also takes care of resuming the prisoner's calling thread after policy evaluation is done, using an OS tracing primitive.

Preloading takes place in the mini jail environment. In this environment, the prisoner is allowed to only make those system calls necessary to initialise the preload library. The ShRO region set up is completely controlled by the jailer, which checks the correctness of the system calls and arguments made during the preload phase. The mini jail environment ensures

that even if the pre-load procedure can be tampered with, no harm can be done. At most, an agent could mess up its own state, but the jailer's enforcement mechanisms are always in place before the prisoner's code is run.

The prisoner's own code is invoked only after after preloading and ShRO setup is complete. Preload techniques can be applied to (custom) dynamic loaders other than the ELF dynamic loader, and alternative techniques (e.g., using binary patching) are conceivable for static binaries also.

The interception layer handles copying of the system call arguments to ShRO, after which it passes control to the policy layer. The policy layer makes a decision on whether to allow or deny the system call depending on the arguments. The policy layer also normalises and expands symbolic links in file name arguments, such that it uses an absolute pathname for comparison with the policy, or for applying pathname substitution when a CHRDIR directive is specified in the policy file. The policy layer informs the interception layer of any changed arguments (e.g., pointing to a new ShRO region where an expanded file name is located) if applicable.

The policy layer does not have to fetch system call arguments, which is to some extent platform specific[35]. It receives the system call arguments in a system-independent (normalised) format from the interception layer. The policy layer bases its decision on the safe copy of the arguments in ShRO, as provided by the interception layer. The policy layer may replace any argument (e.g., by prefix substitution for a virtual changeroot environment) according to the policy. To do so, the policy layer allocates a piece of ShRO memory, writes the substitute argument in this piece of memory, and modifies the original arguments to point to the rewritten argument in ShRO memory.

The interception layer ensures that the safe, possibly rewritten, arguments are used to execute the system call, rather than the original arguments in the prisoner's address space. After the policy layer agrees with the system call (determined by the return value of the policy layer upcall), the interceptor allows the system call to proceed using the possibly modified argument values, possibly after updating some registers in the operating system. How these updates are made is hidden inside the interception layer, as this is operating system specific.

6.4. Implementation

The user-level jailer under Linux is implemented using the architecture outlined above. The modified strace [5] program is the basis for our *ptrace*-based interception layer. Strace is an open-source program that is normally used for debugging purposes to display the system calls that a process makes[36]. Strace provides code for interfacing with a number of tracing systems

[35] *Ptrace* is rather inefficient at reading data from a prisoner, as it allows only one word to be read at a time. However, most operating systems provide more efficient mechanisms (e.g., Linux has a /proc/mem device) for reading from a child process's address space. When this is not the case, a poor-man's /proc/mem device could be implemented by mapping in the address space of a jailed process at fork/exec time using shared memory primitives.

[36] Strace may be replaced by a custom-build, small tracing layer which can be more easily scrutinised on possible security

(such as *ptrace* and */proc*) on different platforms. This can make porting the interception layer to other platforms simpler.

A second interception layer has been implemented for a modified Linux kernel that maintains an in-kernel **action table**. An action table keeps an in-kernel cache of policy decisions for a given jail, such that it can prevent upcalls to the jailer program for system calls without complex arguments that the jailer has made a decision for before. The kernel jailer is implemented as an extra Linux system call. This kernel jailer automatically jails all children of a traced process. System call events are exchanged between the kernel jailer and the user-level jailer program via messages over a System-V message queue.

The kernel jailer provides a mechanism for fetching arguments (registers and de-referenced arguments such as file names) from a traced process's address space using a request/reply mechanism that uses standard System-V IPC *msgsnd/msgrcv* system calls. Except for short arguments (where the System-V message queue is actually more expensive than a *ptrace* call), this is more efficient than the *ptrace* mechanism that reads one word per system call from a traced process's address space. Similar to *ptrace*, the kernel jailer allows for updating system call argument registers in the Linux kernel. A new interceptor layer for integrating the kernel jailer was written; other than that, nothing was changed to the user-level jailing system's code. No policy decisions are hardwired in the interceptor layer: the first time a prisoner makes a particular system call, the call is always passed to the policy layer in the user-level process which makes a decision.

As the kernel does not have to implement a mechanism for securing system call arguments, its implementation can be kept small. The main part of the in-kernel jailer extension consists of 390 lines of code. The Linux *ptrace* based jailer is the focus of the remainder of this section. Performance of the kernel jailer is assessed in Sec. 6.5.

A number of implementation issues in the *ptrace* based interception layer need to be resolved, which are not unique to our system. For example, *ptrace* does not always guarantee that forked children of a prisoner are automatically traced. Linux allows setting a flag on the Linux variant of *fork*, *clone*, which determines that a child is traced. The interception layer sets this flag for each *clone* call by a prisoner. For other systems, a solution described in an earlier paper [49], is used, which consists of placing a breakpoint just after *fork*. This gives the jailer time to attach to the forked process using a *ptrace* primitive, after which the jailer removes the breakpoint and the child continues execution. Note that this is a potentially vulnerable solution, as care must be taken that no process in the same jailer can access the part of the prisoner's address space where the breakpoint resides (e.g., using *mmap*), and remove it prematurely. This can be mediated by suspending other prisoners during execution of a fork. Fortunately, in Linux such issues are avoided as its *clone* call provides a flag that specifies that children of a traced process must also be traced.

A new ShRO region must be preloaded at the time that an *execve* is done. The arguments of the *execve* call are modified such that the loader forces preload of the ShRO

holes than strace, which was not designed for security. However, strace provides mechanisms for system call normalisation which are useful for prototyping.

environment. When a process creates a thread (i.e., calls *clone* on Linux), this thread shares the ShRO region with all other threads of this process, so no further work is required; this also applies to *fork*. The jailer makes sure that ShRO is safe in view of concurrent access by multiple prisoner threads within a single jail.

The policy layer is called by the interception layer using a very simple interface. The policy layer provides a *syscall_pre* and a *syscall_post* method. These methods take a pointer to a normalised *syscall* argument buffer as an argument. This buffer is filled in by the interception layer with a (normalised) copy of the arguments (i.e., the system call's register set in Linux) of the system call. In addition, it contains the (normalised) system call number and the caller's process-ID and thread ID (if applicable). Based on the normalised system call number, the policy layer can deduce the meaning of the arguments of the system call. If required, the policy layer can request the interception layer to de-reference a specific system call argument (e.g., a pointer to a file name or a network address) from the prisoner's address space, which it does using OS-specific mechanisms. If required, checked and possibly modified arguments are frozen in ShRO (Sec. 6.2.3), with the corresponding argument register changed and updated in the kernel by the interception layer.

Fetching arguments and freezing them in ShRO is only done when required, on request by the policy layer, depending on the system call that was made. Whether syscall_pre or syscall_post is called is determined by the interception layer depending on the call that was made, in some cases depending on earlier results of calling the policy layer. For example, the *syscall_pre* method may return a code which indicates that the policy layer is only interested in post-system call notification for this particular system call for future events. Based on the information obtained from the interception layer, the policy layer makes a decision on whether it allows or denies the call, possibly after rewriting the argument.

The *ptrace*-based interception layer maintains an *action table* internally. The policy layer in some cases decides that a system call is always allowed or always denied. It notifies the interception layer by returning an appropriate return value to the interception layer. The *ptrace* based interception layer stores this return value in its internal action table, such that if the same system call is made again, it is immediately allowed or denied, depending on the policy layer's decision. If denied, the policy layer specifies an errno (return value of the system call) to be reported to the prisoner. If the in-kernel tracing mechanism supports this, it can maintain an action table in the operating system, such that the jailer will not have to be notified of all system calls. Some system calls can then be allowed or denied instantly by the kernel, which improves efficiency significantly, as *always* decisions avoid the overhead of switching to the user-level jailer program for these calls. For example, read or write calls are almost always safe, as they use a file descriptor that was returned earlier by a successful verified *open* or similar system call, e.g., *connect* or *accept*[37]. An in-kernel action table is maintained by our Linux in-kernel tracer implementation.

[37] There is an issue with using read or write on an UDP socket, which may have to be checked if the policy specifies a limited set of peers from which datagrams may be received -- then all *recv* and *read* calls on such a socket must be checked by the jailer individually. However, for most policies this will not apply (see Sec. 6.4.1), and *read* and *write* system calls will therefore generally be allowed instantly.

When a system call is denied, the policy layer returns this decision and a normalised error code *(errno*) to the interception layer. Ptrace, however, does not provide a straightforward mechanism for denying a system call [49]. Therefore, the *ptrace* based interception layer executes a harmless *getpid* call instead of the original call, and substitutes this call's return value for the error code specified by the policy layer before resuming the invoking thread. An interception layer that uses a more sophisticated tracing system can typically specify an error code and truly deny the call using a tracer primitive.

6.4.1. Post-system call policy evaluation

The ShRO region provides an efficient and simple security measure for arguments that are specified by a prisoner before making a system call. There are a few system calls for which a potentially policy sensitive argument is only known *after* the system call has been made. This applies in particular to TCP *accept* calls and UDP *recv / recvmsg / recvfrom* primitives: the peer address is only known to the kernel after *accept* or *recvfrom* takes place. The problem is that the result of the call is written into the caller's address space by the kernel, and between the time that the result (e.g., *peeraddr)* is returned and the jailer checks this, another thread of the prisoner could have modified the returned value such that it passes the jailer's policy check. Note that the operating system has already created the file descriptor as the result of executing the *accept* call. For this reason, keeping the invoking thread from resuming until checking is done is not a feasible solution, as the file descriptor can be easily guessed and used by another thread to send out information even before the jailer has had a chance to check the peer's address against its policy.

An approach first introduced in the Ostia system [41] is used to handle this issue. The Ostia approach is based on a *delegation mechanism,* where every sensitive call (such as *open* or *accept)* is executed by the jailer instead of the prisoner. As the jailer is the process that executes the system call, it can be sure that the prisoner has no possibility of modifying the system call arguments or using the file descriptor before a peer's address has been verified. Most sensitive system calls return a file descriptor. The jailer can pass this file descriptor to the prisoner over a UNIX domain socket, after which the prisoner can use it. The authors of the Ostia paper implemented a kernel extension (implemented as a loadable kernel module) to handle transferring the file descriptors from the jailer to the prisoner. This solution is not usable in our system, as it is at odds with our requirement of being able to run on unmodified UNIX systems without system administrator intervention.

In our jailer, post-system call processing is implemented using a *trampoline* construction: when a post-system call routine has to be invoked by the prisoner after a system call has been made, the jailer sets the return address (program counter) of the invoking prisoner thread to an address of a dispatcher routine in the preloaded executable library. When the jailer allows the kernel to proceed with execution of the call, the operating system resumes the calling thread at the modified return address after executing the system call. As a result, the

dispatcher routine is run, which calls an appropriate handler routine, e.g., to receive the file descriptor from the jailer, and then returns to the original prisoner's return address. Note that when the jailer executes the delegated call, the original call of the prisoner must be aborted to make sure that the prisoner does not do an actual (unverified) *accept*. In the *ptrace* based jailer, this is achieved by replacing the original system call number for that of the harmless *getpid* system, call.

The post-syscall processing mechanism makes sure that these mechanisms are completely transparent to the prisoner. The dispatcher routine is written in assembly language to handle certain architecture specific things, such as saving / restoring registers according to the i386 convention. Except for these 20 lines of assembly code, all the jailer code is written in C to be portable.

The trampoline construction is only invoked for a few calls. Two calls are not security critical, but important for consistency. The *readdir* and *getcwd* call's returned directory names need to be modified by the jailer before return to the calling thread, such that the returned directory names are consistent with the virtual change-root environment applied to the prisoner. The *accept* call is invoked in the jailer, that must check the peer's address after the call is made. After that, the jailer passes the file descriptor to the prisoner. This is achieved by letting the preloaded dispatcher routine read the file descriptor from the UNIX socket between the prisoner and the jailer. *Recvmsg* on an UDP socket requires similar handling in the jailer when the policy specifies a limited set of peers that may send datagrams to the prisoner. When a prisoner invokes *recvmsg*, the jailer first *PEEK*s the socket (*PEEK* leaves the datagram in the socket's queue so it can be read again) to check the sender's address, and if not allowed discards the datagram by reading it from the socket and discarding it before reading the next datagram. If a message is found whose sender is allowed, the prisoner's thread is resumed, so it can read the datagram from the socket. The prisoner is unaware of any failed *connect* calls or discarded datagrams.

Listen must also be post-processed, such that the jailer obtains a copy of the allocated file descriptor (via a UNIX domain socket) so that it can later do an *accept* using this file descriptor. The jailer does the necessary bookkeeping in order to correctly handle *read* and *write* (or *recv* and *send*) calls on socket descriptors. In particular, it has to keep track of the type of a socket, i.e., whether it is a TCP socket or an UDP socket to be able to handle calls on these sockets correctly.

Note that the delegation approach as outlined above is only required when the policy specifies that only some senders are allowed to connect or send datagrams to a prisoner. Otherwise, a server process is allowed to accept connections from any party, and processes that are only allowed to communicate with specific parties will typically use connect only and will simply not be allowed to accept. The delegation technique outlined above imposes some extra overhead due to the system calls required for file descriptor passing. It is therefore bypassed in case of a policy that does not specify conditional peer addresses for TCP *accept* or UCP *recvfrom* calls, or which simply refuses *accept* or *recvfrom*.

6.5. Performance

This section shows performance results for both jailing systems implemented, the *ptrace*-based jailer and the kernel jailer. Micro-benchmarks are used to investigate and analyse the overhead of some representative system calls, and present the performance of three applications whose performance is dominated by system calls, so they represent a ‘‘worse case’’ for jailers.

All experiments are conducted on an Athlon 64 3200+ with a Linux 2.6.13.2 kernel, compiled with our kernel jailing patches. The same benchmarks are presented when run outside the jail, and under control of strace [5], a system call tracer that uses *ptrace*, modified to intercept all system calls but to generate no tracing output. The latter comparison exactly shows the overhead incurred by the *ptrace* mechanism, from which the time spent in the jailer can be deduced.

The kernel jailer offers the possibility to do one important optimisation: system calls that are always denied or always allowed and thus need no intervention from the jailer process are handled completely in the kernel by the kernel jailing code.

6.5.1. Microbenchmarks

Table 25 presents the performance of microbenchmarks that each invoke one system call in a tight loop. The time presented is the average time for one system call. Measurements are presented for unjailed benchmarks, benchmarks run under the control of strace with output disabled, and under control of our ptrace and kernel jailers.

Syscall	Unjailed	Ptrace	Ptrace jail	Kernel jail	calls in loop
geteuid	0.07	5.1	6.2	0.08	100000
stat	0.85	7.2	14.0	14.3	10000
getcwd	0.51	6.4	12.7	9.5	10000
accept	91	169	537	466	1000
connect	98	178	508	466	1000

Fig. 25. Microbenchmarks of selected system calls. Time is in µs per system call

Geteuid is a system call that takes no argument and is always allowed by the jailer. This benchmarks shows that the impact of *ptrace* intervention is considerable in comparison to system call times; the overhead is dominated by context switching between the benchmark process, the kernel and the jailer. This benchmark does not require argument fetching or rewriting, or nontrivial policy logic by the jailer; accordingly, the extra time added by the jailing part is small, 1µs. Because this system call is always allowed, the kernel jailer allows it

without consulting the user-space policy engine. Its performance is therefore comparable to the unjailed case.

Stat takes a relative file name ‘‘junk’’ (in the prisoner’s jaildir) as an argument and returns this file’s status. It requires securing of the file name in ShRO and updating the register for this argument in the kernel. The user-level jailer program keeps track of the prisoner’s current working directory, and uses this information to convert the relative pathname to an absolute pathname. The ptrace jailer requires two *ptrace* system calls to retrieve the 5-byte file name from the prisoner, where the kernel jailer requires a *msgsnd* and a *msgrcv* call. However, the latter calls are each more expensive than a *ptrace* system call. The contribution to the overhead of various parts of the jailer is analysed below.

Getcwd returns the prisoner’s current working directory. It requires a rewrite of the returned directory name in the prisoner’s address space using a post-syscall processing routine when the prisoner runs in a change-rooted directory. In this case, only the post-syscall routine is called, but no rewriting is necessary. The ptrace jailer requires seven *ptrace* system calls to retrieve the directory name, whereas the kernel jailer requires only one pair of system calls. This difference makes the kernel jailer faster.

Accept and *connect* show that these calls are relatively expensive. For these benchmarks, a client and a server program are run on the same machine, one in a jail and one free. Each connection setup requires process switches between benchmark processes. *Accept* is done by the jailer, and the resulting descriptor is returned to the prisoner over a Unix domain socket.

jailer	**ptrace**	**kernel**
intercept bookkeeping	1.1	1.1
read syscall args	1.02	2.35
canonicalize args	0.86	0.81
check pathname	0.52	0.54
update kernel registers	0.27	0.58
microtimers, various	0.69	0.54
total jailer costs	**4.46**	**5.93**
basic tracer overhead	6.35	unknown
system call	0.85	0.85
kernel extra	2.34	unknown
total time	**14.0**	**14.3**

Fig. 26. Breakdown of the time spent by the jailer for a stat system call, in microseconds (µs, averaged over 10000 runs.)

Connect also requires freezing the argument in ShRO. Both in the ptrace and the kernel jailer, accept causes more overhead than connect, compared to the unjailed case. The delegation mechanism (Sec. 6.4.1) requires the use of different threads in the jailer. For the ptrace

jailer, the difference between accept and connect is larger than with the kernel jailer. This is attributed to a peculiarity of the *ptrace* implementation, which allows only the main jailer thread to make *ptrace* calls. In contrast, the kernel jailer allows all threads in the jailer to make controlling jailing system calls.

Table 26 shows a breakdown of the various parts of the jailer code for a *stat* system call, measured by nanosecond timers inserted into the jailer code. As for the *getuid* call, bookkeeping in the jailer costs 1.1µs. Reading the file name from the prisoner address space is implemented by reading a word at a time with the *ptrace* system call, and this is the largest of the jailer costs; for the kernel jailer, an even more expensive pair of System-V IPC *msgsnd/msgrcv* calls is done. Calculation of the canonical path name and checking this against the policy paths is also a noticeable contribution. Another *ptrace* or kernel jailer system call is involved in copying the pointer that points to the immutable copy of the file name (in ShRO) into the prisoner's register set. For the ptrace jailer, the time spent in the jailer should equal the difference between a jailed system call and an unjailed, ptraced system call. However, 2.34µs are unexplained. This difference must be attributed to kernel peculiarities: timers indicate that the time the kernel takes to actually perform the system call for the prisoner, is approximately this much longer for a jailed prisoner than for a prisoner that is controlled by strace.

6.5.2. Macrobenchmarks

Three macrobenchmarks were run to measure the overall performance of the jailing system. These macrobenchmarks emphasise different aspects of the jailing system. All macrobenchmarks are non-trivial for a jailing system: they make a relatively large number of system calls compared to the time used for doing computations, as shown by the columns for system time and user time in Fig. 27. Many applications require (far) fewer system calls than the benchmarks presented.

	Unjailed			Ptrace	Ptrace jail		Kernel jail	
	system	user	total	total	total	jailer upcalls	total	jailer upcalls
configure	2.2	3.4	**6.7**	**9.14** (36%)	**14.3** (113%)	367,320	**11.7** (75%)	147,947
make build	3.5	10	**14.6**	**19.0** (30%)	**27.4** (88%)	598,770	**24.0** (64%)	264,815
ant build	1.2	15.5	**16.7**	**22.4** (34%)	**24.5** (47%)	669,557	**18.1** (8%)	62,562

Fig. 27. Results of an *strace* configure script, a *make* build, and a build of a large Java source tree using *ant*. Times in seconds, between brackets the percentages overhead imposed by jailing.

The first macrobenchmark is to run a configure shell script for the strace source code tree. This script executes a number of programs that test availability of required functionality on the operating system. It presents a worst-case scenario for the jailing system: *execve* calls require preloading and setting up a new ShRO region for the new process.

The second macrobenchmark is a build of this jailer system itself using make. To find out dependencies and compile accordingly, make and its spawned subprocesses must open many files and generate many new files. This benchmark is dominated less by system call time than the configure script.

The third macrobenchmark is a Java build system (*ant*) which compiles a large Java source tree. The sources consist of 1005 Java source files of a total length of 181073 lines (5554227 bytes), which are compiled to Java bytecode using the IBM 1.4 Java compiler and virtual machine. Ant is a multithreaded Java program. Considerable time is spent both in compiling the source code and in reading and writing files.

What these benchmarks show is that it is possible to run nontrivial, multithreaded or multiprocess applications within a jail with reasonable performance. The measured applications make a large number of system calls. Despite that, the overhead imposed by the ptrace based jailer is a worst-case of 113%. The columns ‘‘jailer upcalls’’ in table 27 show the number of times that the user-level jailer process is consulted for making a decision in both the ptrace and the kernel jailer. In the ptrace based jailer, the user-level jailer process is consulted for every system call, whereas the kernel jailer consults this process only once for many of the system calls. With configure and make, the kernel jailer is capable of deciding the system call verdict immediately from its action table, without dispatching to the jailer process, for about half the number of system calls made by the prisoner. As a result, the jailing overhead drops to 75% for make for the kernel jailer. A significant part of the jailing overhead in these cases, although more so for configure than for make, is caused by the expensive *execve* call.

The Ant Java build system (a nontrivial Java program) incurs significantly less ptrace jailer overhead (47%). Most of the overhead in this benchmark is caused by *ptrace* overhead. This is due to the fact that the system calls made by Ant are generally less expensive to handle than the system calls made by configure and make. When using the kernel jailer, the user-level jailing program is only consulted for one tenth of all system calls. The majority of system calls that is immediately allowed is for manipulation of thread signal masks, which the tested version of Java does very frequently. As most of the jailing overhead is incurred by switching to the jailer process, the kernel jailer provides significant performance gain compared to the ptrace jailer. In conformance with the relative time spent in user mode and system mode, the total overhead for jailing Ant is much smaller than for the other two benchmarks, in both jailing systems.

The benchmarks shown in this chapter tested ‘‘hard cases’’ for the jailer. For applications that spend much of their time in user mode, better performance is expected than for the benchmarks presented above. Indeed, experiences with some applications (e.g., gzip of large files, results not shown, where the majority of time is spent compressing data in user-mode) confirm the—trivial—assumption that jailing overhead drops to nearly zero for both jailers when the application spends only a tiny fraction of its time in system mode.

6.6. Related work

A number of different designs exist that address system-call interception based jailing of untrusted programs [96,117,49, 41, 106]. A number of systems [96,41,106,93,118] depend on modifications to the operating system to function. Jailing systems that require changes to the operating system have deployment drawbacks, as few system administrators are willing to modify their operating system kernel. *Systrace* [96] is a notable exception in that it has reached significant deployment: *systrace* is part of a number of open-source BSD UNIX systems. *Systrace* requires manual policy generation for every program. It is primarily aimed at generating policies for known programs, such that these cannot exceed their normally required permissions, for example, after an intrusion took place. However, like most jailing systems, *systrace* cannot deal effectively with confinement issues due to the lack of a jailing model that deals with runtime-determined system call arguments such as IPC tokens, as explained in Sec. 6.2.

A number of system call interception based jailing systems [8,117,49] run on unmodified UNIX systems using standard debugging support such as *ptrace* or */proc*. However, these systems suffer from a number of race conditions that rendered them insecure for many modern (e.g., multithreaded) applications [39]. Consh [8] provides a virtualized environment for applications. In consh, an untrusted application's resources (e.g., file system) are mapped onto local or remote resources, giving the user some control over, for example, the directories that a prisoner can be access. Consh is based on the original Janus [117] */proc* based jailer, and as such it suffers from the race conditions that made Janus and other */proc* or *ptrace* based jailing systems insecure. None of the above described jailing systems provide a model that deals with confinement and runtime argument based policy decision issues as our system does. Liang et al. [62] describe an approach for isolating effects (in particular, on the file system) of executing untrusted programs on standard Linux; their implementation uses the stack relocation approach discussed in Sec. 6.2.

An alternative to system-call level jailing is language-based sandboxing such as provided by, for example, Java or Safe-Tcl [87]. Compared to language-based systems, system call interception based jailing has the important advantage of being language-independent. Also, language-based sandboxing systems are complex: getting a language's security model right is far from easy [15, 123]. System call interception is independent of, and can effectively safeguard from vulnerabilities in, any language's security enforcement mechanism.

Various operating system level techniques or new operating systems have been proposed to achieve better security, flexibility, or software fault isolation for different concurrently executing applications. Notable examples are [42,33,64,31]. In addition, operating system security enhancements are proposed or implemented that support enforcement of mandatory access control policies, such as DTE [122] or Security Enhanced Linux [4]. Several of these designs could increase security or software fault isolation. Contrary to these approaches, our jailing system supports secure confinement of applications on top of existing, standard UNIX platforms.

Other current approaches for securing systems are based on operating system visualisation or virtual machines. For example, Solaris containers [67] are virtualized instances of the Solaris operating system within this operating system; FreeBSD jail [53] provides a similar, near-complete virtual instance of the FreeBSD operating system. VMware [6] or Xen virtualization [30] can be used to run multiple operating systems concurrently on a single machine. Virtual machine approaches are relatively heavy-weight as a full operating system instance runs in each virtual machine; for this reason, virtualization is not suitable for isolating a very large number of concurrently executing programs individually. Also, it is unclear how information flow can be controlled between different virtual machines or operating systems.

A number of (recent) host protection systems focus on software fault isolation (SFI [66,121]) techniques to protect against software vulnerabilities [37,89,90]. These approaches use binary translation [66,121] to dynamically rewrite all potentially vulnerable parts of a program that could lead to exploitation of vulnerabilities (e.g., using code injection) such as memory writes, jumps to potentially malicious code, etc., to safe code that runs in a protected part of the program that cannot be reached by an attacker. The translated code securely guards all sensitive operations. A common attack scenario used to guide SFI designs is where an initially trusted program (sometimes securely loaded, sometimes privileged) is compromised by an external attacker; techniques exist to prevent these types of attacks. Current SFI systems often do not come with a system call argument based jailing model or policy[38]. If jailing policies are used at all, these are typically rather simple and must be configured by hand [89].

Process firewalls [118] present an approach where certain attacks can be prevented automatically using a ‘‘firewall’’ implemented in the kernel, that can detect common attack patterns such as TOCTOU race conditions on file names. The advantage is that these patterns are detected system-wide, irrespective of what user or process invoked a call: any call that may lead to a TOCTOU vulnerability in another process can be stopped automatically. The approach requires modification of the operating system.

Similar the systems indicated above, the process firewall does not come with a jailing model that automatically limits the scope of what prisoners are allowed to do. In a mobile agent system, safe defaults and a strong jailing model are needed to protect the user who executes Mansion, as well as other agents, against potentially malicious agents. The Mansion jailer comes with such a model. The jailing model works for most programs without modification, providing safe defaults and including dynamic system call arguments that are not known at policy definition time. The Mansion jailer never trusts the prisoner. It may be possible to use SFI techniques to augment the jailing approach or vice versa; either way, the Mansion jailer provides a safe user-mode fallback that can withstand TOCTOU races to safely evaluate system call argument policies in all cases.

[38] Another disadvantage of using current binary translation approaches in mobile agent systems, is that these do not readily support self-generating code, as used in for example JIT compilers [90]. This is a disadvantage when using mobile agents, which for portability reasons are often written in Java.

The Mansion jailer distinguishes itself from existing (jailing) systems by providing a clear jailing model that defines precisely what a jailed process can and cannot do, both inside and outside the jail. Making a distinction between a prisoner's allowed actions within a jail and outside, the jailer allows complex multithreaded programs, or multiple programs that use IPC, to be executed unmodified in a jail, while these programs' interactions with the outside world can be tightly constrained.

6.7. Conclusion

The jailing system presented in this chapter provides a simple but effective jailing model. Our solution is the first that presents an effective and secure solution for alleviating shared memory and file system race conditions, without requiring special in-kernel support for securing system call arguments. This solution is based on copying sensitive system call arguments to a user-level shared memory region to which the prisoner only has read-only access, before allowing the system call to continue. This solution is portable to any (POSIX compliant) UNIX system, even if it has only rudimentary system call tracing support such as provided by the *ptrace* or */proc* system call tracing interface. The main advantage is that the jailer allows execution of untrusted programs in a secure way without any changes to the operating system, and without system administrator intervention.

The jailer adds unavoidable performance overhead to system calls that require evaluation of its arguments by the user-level jailer program, and thus require a context switch to this jailer program before the system call can proceed. By using a kernel jailer with an in-kernel action table, upcalls to the user-level jailing program can be avoided for many system calls (such as read and write), and measurements of the kernel jailer based implementation indeed show a significant performance improvement compared to the ptrace based jailer for the programs that we tested. In the case of the ptrace jailer, the jailing costs are comparable to the basic system call tracing (*ptrace*) overhead for the majority of system calls, although a few system calls (such as execve and open) cause more jailing overhead. Using *ptrace*, a worst-case program (configure) causes 113% compared to not jailing this program; a less system-call intensive but non-trivial Java program causes 47% overhead.

The jailing model provides hooks to prevent information flow between untrusted processes, yet allows processes within the jail to use regular IPC mechanisms and signals in the normal way. Actions that influence the outside world are guarded by a simple, user-defined policy. Policy modification, in particular to adapt the policy to the local system's directory structure, is straightforward, and generally required only once. The jailer can impose strong confinement on programs to protect systems against malicious agents, yet it allows modern programs to run in a jail directly, without any modification.

Future work on using the jailer for resource management will have to demonstrate whether constraining system time by managing system calls is feasible in all cases, but initial experiments show promising results (App. 3).

Chapter 7
Objects and the Mansion object server

This chapter explains the design and implementation of the Mansion object server (MOS). The MOS is one of the most important components of the middleware. Although invisible to agents, all objects, including the room monitor object, are run in a MOS. Many services (such as the location service) are implemented as objects that run in a MOS. The Mansion (object) security model includes zone-based access control policies.

The MOS is written in C/C++, and currently supports objects implemented in C++. The MOS IDL and data representation (Sec. 5.2.3) are designed to be portable. Implementing stubs for agents in different languages, such as C/C++, Java, Python, etc., is straightforward. The security model is based on per-object ScID-based access control lists to constrain access to Mansion objects, and general zone-based policies that restrict access to the MOS (Sec. 8.2.10). The MOS is designed as a stand-alone component that hosts nonreplicated objects. Possible mechanisms for object replication are described in a future work section.

7.1. Requirements

Mansion objects are remotely invokeable objects, hosted in a stand-alone object server. The implementation of a Mansion object is currently tied to the MOS. The requirements of the Mansion object model and the Mansion object server are described in this section.

- The object server should be simple. Complex designs lead to errors and vulnerabilities. There is no need for an object server to be complex. All it has to do is host objects. Currently, objects are not replicated, but there may be multiple MOSes in a zone each hosting different objects.
- A MOS should be able to host many objects, as it may host many objects in a zone; an object management interface should allow creation and deletion of objects in a MOS.

- A MOS should support persistent objects, i.e, objects that can survive a crash or restart of the object server.
- Like AOS, the MOS is not distributed, and it should not depend on external services. (In principle, registering objects in a location service is the responsibility of the client program and not of the object server, to avoid dependencies).
- A MOS should use the same RPC system as other components in the middleware. The invocation and marshalling system should be portable and usable from stubs in different languages; the requirements are similar as outlined in chapter 5.
- A MOS should use ZAC and be able to operate on top of regular TCP connections as well as AOS; in both cases, the client process should be authenticated using the ScID-based authentication protocol (Sec. 3.2.1).
- Mansion objects should have clear ownership, and at least support simple ScID-based access control lists (ACLs) that define access on a per-method basis.
- The object server should support the zone-based authentication model outlined in Sec. 8.2.10. Within a zone, a MOS trusts the client middleware (MMW) to pass an appropriate ScID (e.g., AgentOwnerID) to it, which is passed to the object. For world-wide services reachable from outside the zone, a MOS authenticates the client process and passes the client's ScID or ZoneID to the object.
- A MOS should be able to jail certain objects, or certain object types (classes). This to protect the system or the user executing a MOS against untrusted, malicious or faulty or vulnerable object implementations.

A MOS is a simple infrastructure to instantiate and run hosted objects, and to invoke them. Client programs create objects, and are responsible for registering the object's contact address in a location service.

A MOS uses the same ZAC/RPC system as other Mansion components; the MOS interface for creating and invoking objects, as well as the individual objects, have Mansion Contact Records (MCRs, Sec. 5.1.3), which can be registered in the Mansion location service. Client programs connect to a MOS over a (zone) authenticated channel. If a client program is identified as belonging to the same zone as a MOS, it accepts that a client middleware process (which is then trusted) passes a ScID called *AgentOwnerID* with each invocation. This ScID is passed to the object and is used by it to check its access control list (ACL). Depending on configuration, a client program outside the MOS'es zone may be allowed to connect, for example if a MOS hosts Mansion *services*. In this case, the connecting program is not trusted: the ScID (ZoneID) of the connecting program passed to the object to check the ACL. Note that in both cases, the object sees a ScID; the access control mechanism using ACLs is collapsing both types of ScID—whether a ScID identifies a zone, or an agent/user—in both cases, the object sees a ScID. ACLs are defined per object, by the object's owner.

Fig. 28 depicts an example distribution of MOS and middleware processes on different machines in an single zone. Services such as the object location service are not shown. Each Mansion object server can host a number of objects.

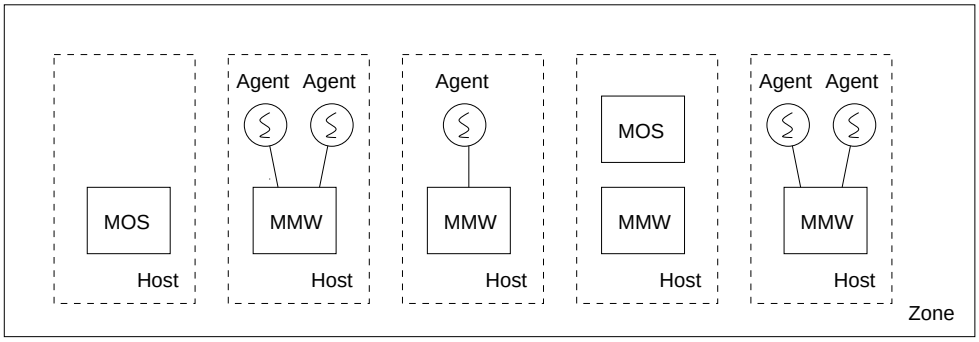

Fig. 28. A set of MMW processes (hosting agents), and two Mansion object servers in a zone, distributed over 5 hosts.

In Fig. 28, agents are hosted on different hosts than the object servers. Two object servers are shown, which each may contain different objects—RMOs of rooms in the zone, or regular objects containing files such as images that agents can search.

In Mansion, contact addresses of objects are registered in the location service using a location-independent handle (Sec. 3.3.4). The handle contains the ZoneID of the object. With each object handle, multiple contact addresses can be registered, one for each object instance. As a result, object replicas in different MOSes can be registered under a single name.

Mansion does not provide mechanisms to replicate state or invocations automatically as part of its object model. Although zones can in theory be geographically distributed widely, it is unlikely that this will occur in practice. Agents are shipped to objects (rooms), making it possible to limit the geographical distance between (a replica of) an object and the agents that use it. This in contrast to, for example, the Globe distributed object system [107] where a driving use case is that clients may be located anywhere in the world. In such cases, wide-area replication of objects can be a solution [17], for scalability problems that occur in this context. Note that for objects that are often updated (e.g., the RMO) and which serve only limited amounts of data (e.g., a few attribute sets) to closely located clients (e.g., on a LAN), as may often be the case in Mansion, having a single object running on a high-performance machine may be a better solution than object replication.

7.2. The Mansion object model

Mansion objects are remote objects, which can be invoked by client processes or by Mansion agents. All Mansion objects are hosted by a Mansion object server process, or by more than one MOS if the object is replicated.

An object is a passive entity whose state may be read or manipulated using remote method invocations. An object contains data (state) which can be accessed through its interface, by agents or possibly by a middleware process, for example, in case of a middleware adding a record to an RMO when the agent enters a room.

Mansion is object-based. Objects are passive software entities, that is, only responsive to method invocations. Objects in Mansion reside in one or more (in case of a replicated object) object server processes. Mansion objects are currently implemented in C++, but can also be implemented in other languages such as C or Java. They are called objects because they encapsulate state behind an interface Objects cannot invoke methods on other objects.

All Mansion objects implement a set of methods defined using the Mansion IDL (Sec. 7.3). All objects inherit a mandatory interface containing a set of methods that implement basic access control related primitives; an implementation (in C++) is also provided. This interface is termed *MansionObject*. Conceptually, all objects in Mansion can be seen as inheriting the *MansionObject* interface.

The *MansionObject* methods are not directly visible or usable by agents; instead, they are used, in collaboration with the MMW, to control access to a given object by certain agents. This is described in detail in Sec. 7.4.

Mansion provides a default *MansionObject* implementation as a C++ object, which all Mansion object implementations can use by means of C++ object inheritance [109]. Objects are however free to replace the default *MansionObject* C++ object by another implementation—or to not make use of object inheritance internally.

7.3. The interface definition language

Mansion has an IDL compiler for objects, that extends and uses the RPC IDL and XDR compiler (Sec. 5.2.3). The syntax of the IDL resembles the C/C++ programming language, with some notable differences, in particular the use of *in/out* and *inout* markers for parameters, similar to the Mansion RPC system.

The IDL uses the basic data types supported by the RPC/XDR system (e.g., *int32*, *char*, opaque character *strings* and *arrays* of primitive types). Constants (*const*) and enumerations (*enum*) can also be defined. A *struct* is used to construct a composite type. An *interface* contains the interface definition, which consists of a grouping of method definitions with arguments and return value. Arguments may be primitive types, constants, enums, or structs. It is possible to include IDL files containing struct definitions, constants or enums.

As an example, the IDL of the *MansionObject* interface is shown in Fig. 29:

```
include "zac/aos/aos.x" // includes SHA1_SIZE definition
                        // from RPC IDL file aos.x

const MANSION_OBJECT_ROLE_SIZE 16;
```

```
struct mmw_role_t {
   char[MANSION_OBJECT_ROLE_SIZE] bitmap;
}

struct mmw_scid_t {
   char[SHA1_SIZE] scid;
}

struct mmw_aclent_t {
    mmw_scid_t principal;
    mmw_role_t role;
}

interface MansionObject {
  int init(in: mmw_oh_t handle);
  int reset();
  int reset_acl();
  int set_restartable();
  int delete_me();
  int acl_get_role(in: mmw_scid_t who,
                   out: mmw_role_t role);
  int acl_set_role(in: mmw_scid_t who,
                   in: mmw_role_t role);
  int acl_get_aclents (in: int offset,
                       in: int count,
                       out: mmw_aclent_t[count] aclents);
  int delete_principal(in: mmw_scid_t who);
  int ping (in: int dummy);
}
```

Fig. 29. The *MansionObject* interface defined using the Mansion IDL

The IDL generates server-side skeleton objects in C++, as well as client-side stubs in C and C++. In addition, the RPC/XDR compiler generates the marshalling routines which used by the stubs and skeletons to pack and unpack arguments. The semantics of the *MansionObject* methods are explained in the following section.

The IDL differs from the original SunRPC IDL, in that *in* and *out* markers are used in the method definitions, rather than different structs for *in* and *out* parameters as is the case for standard RPC. *in* parameters are sent from the client to the server; *out* parameters are returned from server to client, and *inout* parameters are passed to the server with the return value placed at or copied to the same location, similar as explained in Sec. 5.2.3. In the generated C/C++ stubs, *out* and *inout* parameters are passed by reference at the client side. The return value of normal method invocations is a positive value or 0; negative values indicate an error, with specific error codes defined mansion-wide for connection errors or RPC, MOS (invocation), or other specific errors, including authorisation failure.

The method *acl_get_aclents* has an in parameter *count* that is used as part of a subsequent argument definition in the same method. This is a novelty compared to SunRPC, allowing the invoking program to define the size of a variable-sized array at invocation time. Array size arguments are parsed by the IDL compiler and can be either constants, or arguments of

the same method, as shown in Fig. 29. Note that this extension mandates that the object that implements the method defines the semantics of its arguments clearly: for example, if a client program specifies an overly-large *count* or if insufficient *aclents* (ACL entries) are available, the number of available ACL entries up to an object-defined maximum can be returned. The *int* return value of a method is typically used to indicate the number of returned data, aclents here, to the caller, or a negative error code.

The IDL compiler makes use of the XDR compiler to generate marshalling routines. When applicable, the generated code allocates sufficient memory for marshalling/unmarshalling data to or from XDR, if applicable using an argument's runtime value. Internally, there is always a predefined maximum of allocateable data, and memory allocation (e.g., using *malloc*) may also fail; in that case, the method invocation returns an error.

7.4. MansionObject functionality and access control model

Access to Mansion objects is governed by a straightforward access control mechanism. Every object has an **access control list (ACL)**. An ACL allows an object's owner to define which methods may be invoked by what *principal*. A principal is an entity that invokes a method (e.g., an agent), or the person behind that entity. For example, an agent owner. A principal may also be a zone or the zone owner. All principals are identified using a ScID. Unless specified otherwise, principals in Mansion are agent owners, object/room owners, or zones or zone owners[39]. Initially, an ACL is created with only an entry for the object's owner. An object's owner (the principal who created the object) may invoke all methods of the object. Initially, only an object's owner may invoke methods on an object.

An ACL is based on self-certifying identifiers. It contains a ScID for each principal who is allowed to invoke methods on the object. Associated with each ScID is a set of bits, called the **role bitmap**, that defines what object methods may be invoked by the corresponding principal (ScID). The role bitmap takes the order of the methods in the IDL, including the *MansionObject* methods: *MansionObject::init* is bit 0 in the bitmap. A *default* ACL entry (a zeroed ScID) can be created that specifies the allowed methods (if any) for principals who do not have an explicit entry in the ACL. In most cases, it will be straightforward to define a default ACL, which can be set every time an object is created[40]. For example, the default role bitmap for a *MultiFileContainer* object will allow read methods to everyone, but disallow methods that write state into the object.

MansionObject implements the above access control mechanism. The *MansionObject* methods mostly deal with the access control list. Every Mansion object must implement the *MansionObject* interface; this is needed because ACL methods (i.e., for obtaining a principal's or the default role) are called for every invocation. Apart for the object implementing

[39] An agent acts on behalf of its owner; normally, the agent owner's ScID is encoded in the agent's AC at world entrance time. This AgentOwnerID is passed to the object by the agent's MMW, which finds it in the agent's AC. This is described in Sec. 8.2.10 and is described in more detail later in this chapter. ZoneIDs usually do not map directly to their owner but instead are bound to an owner or the organisation that manages them via the zone list in the basement.

[40] A utility program for creating objects does this for *RMO, FileContainer,* and *MFC (MultiFileContainer)* objects.

access control, the *MansionObject* interface is used by the MOS to check status or implement management tasks such as registering an object as a persistent object or deleting it.

The ACL related methods on *MansionObject* are used to define coarse-grained, per method access control. By default, an object checks the ACL for every method invocation, and returns an *access denied* error if the method invocation is not allowed for the invoking principal (ScID). If required, objects can implement additional, more fine-grained authorisation based on ScIDs internally, based on the ScID (of the caller) which is passed with *every* method: the object skeleton generated from the IDL contains a ScID as the first argument of every method. This ScID is filled in by the MOS before invoking the method, using the convention described in Sec. 8.2.10 and 7.1.

Fig. 30 shows some important methods of the *MansionObject* interface, equivalent to the IDL definition shown in Fig. 29. A notable difference between the IDL and Fig. 30 is that in the IDL definition, the first argument *scid* is omitted. This is done for brevity—the IDL definition shows the interface from the client's side. Fig. 30 shows the same interface from the server side. At the server side, an argument *scid_t scid* is automatically added as the first argument of *every* method, to contain the ScID of the invoking principal, as outlined above. Internally, the ScID argument is always there: the IDL compiler generates the ScID field for all internal XDR messages. In some cases, the ScID field is hidden at the interface level. For example, the first argument ScID is suppressed in an agent's stub, because the agent itself cannot modify the ScID argument. But it is present since it is filled in (overwritten) by the MMW process when the XDR message passes through the MMW as a result of invoking a method (Sec. 7.4.1). The ScID field is necessary in the wire representation so that a MMW process can fill in the AgentOwnerID of the invoking agent in the message which is passed to the MOS. The object uses this ScID for access control.

Method	Arguments
int init	in: scid_t scid, in: oh_t object_handle
int reset	in: scid_t scid
int reset_acl	in: scid_t scid
int set_restartable	in: scid_t scid
int delete_me	in: scid_t scid
int acl_set_role	in: scid_t scid, in: scid_t who, in: role_t role
int acl_get_role	in: scid_t scid, in: scid_t who, out: role_t role
int get_aclents	in: scid_t scid, in: int offset, in: n, out: aclent_t[n] aclents
int delete_principal	in: scid_t scid, in: scid_t who
int ping	in: scid_t scid

Fig. 30. Selected *MansionObject* calls

Before explaining the methods shown in Fig. 30 in detail, this section describes how access control lists (ACLs) are constructed and checked at invocation time.

7.4.1. ACL implementation

An ACL consists of a list of entries with the following format:

```
<ScID> <Role bitmap>
```

ScID is the self-certifying identifier of a principal, for example, an agent owner, or a ZoneID depending on the party which makes the call. An ACL entry with a zeroed ScID is the *default* entry. The default role specifies the methods that unknown principals may invoke.

Role bitmap is a bitmap with a bit per method of this object, corresponding to the methods in the object's IDL definition. If a bit is set to 1, the principal named by *ScID* may invoke the method. If the bit is set to 0, the principal is not allowed to invoke the method. Bits have the same order as the order in which methods are defined in the object's IDL definition (see Sec. 7.3), starting with 10 bits for the *MansionObject* methods.

MansionObject methods are included in the object's role bitmap. An administrative program is provided by Mansion for setting role bitmaps for principals in objects, including the default role, which uses the *acl_set_role* call. By default, the administrative program does not enable the *MansionObject* methods in the role bitmap when creating new roles. An object's owner can enable these methods if he/she creates an administrative role.

Initially, when an object is created, only the object's owner (creator) may invoke methods on the object. The object owner's ScID is passed to an object when it is created; at this time, the object's *init* method is called by the object server: the object instantiates the object and passes the invoker's ScID and the object handle to it by calling the *init* method. The *init* method initialises *MansionObject*'s internal ACL data structures. The first-created owner role bitmap has all bits set to 1. This allows the object's owner to invoke all methods. No other ACL entries exist: if another principal needs access, the object's owner needs to create a new (default) ACL entry using a *MansionObject* method.

The first argument of each Mansion object method is a ScID. This argument is passed to the object by the object server, and contains the ScID of the invoking principal. This principal may be an agent (owner) or a middleware process (a zone member), as explained above. Using this ScID, the object looks up the corresponding ACL entry to find out if the principal has permission to invoke a given method. If the invoking principal is not known to the object, the bitmap of the ''default'' ACL entry is used to check permissions. If no matching (default) ACL entry is defined in the object, access is denied.

In the current implementation, checking the ACL using the *MansionObject* interface is done by the generated skeleton code before invoking the method on the actual object. There-

fore, the object need not use the ScID argument or check the ACL itself[41].

Below, an example role bitmap-based ACL for a 5-method object is shown. Besides its own methods, every object inherits the 10 standard methods of the *MansionObject* interface; these are the first 10 bits of the role bitmap, giving a total of 15 bits.

```
4sv5wd4hz0ksjv5wd4n1344sv5wd4hdf:111111111111111 // owner do all
jn1344sv5wd4hz0i0lksjdkljfh742hy:000000010010100
4dw4hzksjdkz0lksjdk5wd4hv54n1dwp:000000010010110
0:                               000000010010000 // default
```

Shown are an administrative role bitmap (all 1's), a default bitmap (some methods allowed), and two specific ScIDs which are allowed to invoke additional methods compared to the default role. Note that except for the administrative role of the object's owner, no principal is allowed to invoke *MansionObject* methods, except for the *get_aclents* call.

The current MOS provides a standard implementation of the *MansionObject* in C++. This implementation is linked with and used by the generated skeleton object, that consists of generated C++ code. The skeleton currently contains unmarshalling code, ACL checking code, the code needed to invoke the method of the object, and code for marshalling the out arguments and return value. Using the IDL definition, the IDL compiler generates a C++ object which contains (initially empty) object code, which should be filled in by the object implementer. After compilation and linking, the object is ready to use.

Concretely: the generated skeleton code receives a marshalled invocation from the MOS, and calls the relevant XDR functions to unmarshall the call. Then, the skeleton invokes a library function generated by the skeleton compiler, called *allowed.* Allowed checks if the caller may make the call by obtaining and checking the principal's role bitmap (or, if applicable the default role bitmap). If allowed, the skeleton invokes the method. This means that an object which uses the standard *MansionObject* access control mechanism need not concern itself with access control, as the skeleton checks the ACL before invoking an object's method.

When forwarding a method invocation by an agent to the MOS, the MMW can fill in the invoking agent's AgentOwnerID in the first (ScID) field of the marshalled XDR message. Internally, this is done by modifying the first (ScID) argument of the marshalled invocation that is passed to the MMW when an agent invokes the object's client stub. This works because, by design, the first ScID argument is always there[42], and because it is a fixed-size uninterpreted byte string (20 bytes).

The marshalled invocation with the modified ScID is forwarded to the MOS and onward to the skeleton, which uses it for access control before invoking the object's method. Note that if the object server runs in service mode (see above and Sec. 8.2.10), the MOS does not trust the information in the ScID field, and modifies it to contain the ZoneID of the invoking process before passing the marshalled invocation to the skeleton.

[41] If required or useful, per-method access control logic can be implemented inside the object, based on the ScID argument passed to each method. This way, an object may implement more fine-grained access control.

[42] The RPC forwarding service (Sec. 8.2.11) checks that an incoming method invocation has sufficient size.

Note that there are alternative uses for the ScID argument besides simple method-based access control. For example, a variant of the *MultiFileContainer* object called *OwnedMFC* has been implemented, in which an agent can view and store data in a private directory. Agents can only see or modify files in a directory that corresponds to their *AgentOwnerID*. The implementation makes use of the ScID by interpreting all requests relative to this ScID, which internally maps on the name of a directory. If no directory for the ScID exists, an error is returned. This variant of the MFC is currently used to implement the Morgue in which returned agents of different agent owners are stored.

7.4.2. The MansionObject interface

This section describes the *MansionObject* methods shown in Fig. 30.

Init is automatically called by the MOS after instantiating the object. It is used to instantiate the internal data structures of *MansionObject*. It takes the caller's ScID, and sets the role bitmap of this ScID to all 1's, to indicate that the creating principal is the owner of the object. Typically, objects are created by a zone administrator, whom sets the invoker ScID to ZoneID so that the *zone* owns all objects in a zone. Normally, (depending on a MOS startup flag) a zone member program manages (create or delete, or define ACL's) objects.

An object handle is also passed to the *init* method. An object is typically given a dedicated directory by the MOS but, depending on MOS configuration, objects may also share a directory structure. The object handle argument allows *MansionObject* to create a directory to store the object's state in. As object handles are unique (only one object with a given handle may run in a MOS), each object has its own directory. This directory can be used by the object implementer to store persistent object data. MOS directories remain on disk, so an object's persistent state can be recovered after a MOS crash or restart. *MansionObject* also creates a separate directory containing the object handle, in which it stores the object's ACL state so that this state can also be recovered after a restart.

Related to *init* is the *set_restartable* method; this method can be used to trigger an object to call back on the MOS using a run-time interface (see Sec. 7.5), to indicate to the MOS knows that it is restartable. It also results in a bit set in the MOS. If the restartable bit is not set for a given object, any left-over state of the object (typically, a private per-object directory) are removed at startup time. The call on the runtime interface ensures that the object's data (which is stored in a per-object directory structure) is retained when the MOS is shut down or restarted. The *set_restartable* method is normally called by the object's owner at object creation time, but the object's implementation may also itself indicate that it is restartable by calling the runtime interface. The way an object serialises its state to disk (e.g., using an object-specific directory structure) and recovers it after a restart, is up to the object implementer.

Restarting objects is not done automatically by the MOS; this should be done explicitly by the object owner. The client program used to create objects outputs the names (object

handles) after creation, which can be stored on disk. These can be used to invoke a *restart_object* method on the MOS for all objects. This returns a contact record for the object, which should next be registered in the location service; the contact record may differ from one created before for a given object. Mansion comes with a ''restart-all'' script that allows a zone owner to recreate all of a zone's rooms and objects—which are kept track of when being created—automatically if needed.

The *acl_set_role* and *acl_get_role* are important methods for ACL management. *acl_set_role* allows the object's owner to associate a role bitmap for a specific ScID. Object administrators can define new ACL entries using the *acl_set_role* method. The implementation checks the current ACL to avoid double entries for a given ScID. *acl_get_role* returns a specific role bitmap for a given ScID. This call is used by the *allowed* call of the skeleton which checks the ACL before invoking a method. The MMW may also invoke *acl_get_role* to check whether a given agent is allowed to invoke methods on the room monitor object at all, before accepting it.

Get_aclents allows an authorised principal to obtain all ACL entries (a list of ScID—role bitmap combinations) from the object. *Delete_principal* and *reset_acl* are used to delete a specific ACL entry by specifying it's ScID; *reset_acl* removes all ACL entries (except for the object owner's entry).

Delete_me can be used to let the object self-destruct. This is the only way to delete an object. Having the *delete_me* call be part of *MansionObject* instead of implementing a *delete_object* call directly as part of the MOS's object management interface (Sec. 7.5), has the advantage that the object's ACL can be used to determine whether a principal is allowed to delete the object. Typically only the object's owner (its creator) may delete an object, but it is conceivable that an object owner creates an ACL for another principal in an administrative role. This can be achieved if the role bitmap has the administrative *MansionObject delete_me* method enabled. An object can, technically, not self-destruct. To delete itself (as the result of *delete_me* being called by its owner), an object has to invoke a call on a runtime interface that the MOS provides, so that the MOS can first clean up any (persistent) state of the object; next, the MOS invokes a mechanism to delete the object (Sec. 7.5).

Finally, the *ping* method simply returns 0 as fast as possible. Ping can be used to measure round-trip times to the object. It can also be used to verify that a given is still alive. If required, access to *ping* directly by agents can be denied by setting its bit in the role bitmap to 0. It is useful to allow zone member (MMW) processes to invoke ping, so they check availability of the object (or a given replica of an object) as part of binding (Sec. 2.3.10).

7.5. MOS layering

The Mansion Object Server internally consists of two layers. One layer, called the **Object Management (OM) layer**, provides an RPC service for managing and invoking objects which is accessible from the outside world. The second layer is internal to the MOS, and

takes care of the actual instantiation of objects (currently, C++ objects) and of invocation of their methods. This layer is called the **Object Instantiation and Invocation (OII) layer** (pronounced as **"O2"**). The OII layer can (dynamically) link in precompiled libraries that contain object implementations (including generated skeletons). The OII is started by the OM layer as a program every time a new object, or an object of a new type, is created. (Various ways in which the OII can be started, are described in Sec. 7.5.3). The OII program provides an RPC interface to the OM using which objects can be instantiated and invoked. The OII also provides objects it hosts with a runtime interface. This runtime interface currently contains a *delete_me* call, which causes deletion of the object and all its state, and a *restartable* call that sets the restartable bit in the OM layer as explained in the previous section. The OM and OII layers are explained below.

7.5.1. The Object Management (OM) layer

The main MOS process implements the layer that takes care of object management, that is, it provides an RPC interface using which client processes can create or delete objects. The object management (OM) interface is the main entry point of the MOS, and it contains three methods, *create_obj*, *restart_obj*, and *invoke*. The signatures of *create_obj* and *restart_obj* are identical; only *create_obj* is discussed here. The invoke method allows a client to ship marshalled invocations to the MOS to invoke a method on an object.

The method for creating objects is *create_obj(object_handle, *mcr)*. Each object is identified by its object handle. This handle must be constructed before creating the object (or the object replica) in the MOS. A handle is constructed using the MOS'es ZoneID, the object's type identifier, and the room number and the number of the object in the room (Sec. 3.3.4). For a service, a handle can be constructed in a similar way. At object creation time, the program which creates the object passes the object's handle to the *create_obj* call.

The object handle is used by the OM to determine the *type* of the object at creation time; this is done using the type identifier which is part of the object handle. The OM layer knows what object type (or *class*, in C++ terms) to instantiate; it can instantiate the appropriate OII. A library is present for every object type, in a local repository of the MOS. This library contains the generated skeleton code for the object and the object code in compiled form, currently statically linked to an OII (Sec. 7.5.3).

The call-by-reference argument **mcr* returns the MCR of the object created to the caller; this MCR can be stored in a location service. To delete an object, it suffices to invoke the object's *delete_me* call. Before deletion takes place, the object first checks the ACL to see whether the invoking principal is allowed to invoke the *delete_me* call. Objects may also do an internal check on whether the invoking principal's ScID is the same as the ScID which created the object (the ScID of the creating principal is stored in a field in *MansionObject*, together with the object handle, by *init*), before deleting itself.

Besides the management interface, the MOS also provides an *invoke* method for invoking objects; currently, this method is provided by a dedicated RPC service. For every created object, the MOS provides an invocation forwarding service, which listens at the communication endpoint returned by the *create_obj* call. This communication endpoint may be created over AOS (if the MOS uses AOS), or as a TCP endpoint; typically, a single TCP endpoint is used for the invocation forwarding service, where the *index* field in the MCR is used to indicate the object to which the invocation should be routed. Internally, the invocation is routed to the OII in which the object resides (see Sec. 7.5.2).

The invocation endpoint of an object is indicated by the MCR returned by the *create_obj* call. This MCR indicates the invocation forwarding service. The object's MCR is used by the MMW to bind to the object—which is effectively a binding to the invocation forwarding service. The invocation forwarding service will route incoming marshalled requests to the appropriate object's OII using the index field from the MCR, where the invocation is passed to the object's skeleton. The skeleton unmarshalls the invocation, invokes the object, and marshalls and returns the result.

7.5.2. The Object Instantiation and Invocation (OII) layer

The OII takes care of managing and instantiating objects, and contains the object type's(linked in) skeleton/object code. The OII provides an RPC interface to the OM layer with which it allows the OM to instantiate or delete objects.

Multiple objects (of a given type) may be instantiated in the OII's address space. The OII *create_object* call takes a class identifier as an argument, which corresponds to the object's type. Each instantiated object has an identifier (an *int*) which is returned to the OM by the OII's *create_object* call. This integer is passed with every operation to indicate which object an invocation is intended for.

The OII is responsible for passing marshalled invocations to the appropriate skeleton routine at invocation time, which which unmarshalls the request and invokes the method on the instantiated object. After return of the method, the skeleton marshalls the return value and *out* arguments and sends the result back to the OM layer, which ships it back to the caller.

7.5.3. OII management

The OII is a program, started up by the OM. This way, flexibility is provided to the object server on how it manages and runs objects. Depending how objects (OII's) are started, objects can be completely isolated from each other (a single object per OII), or isolated per type (all objects of a given class are run in one OII), or not isolated at all (every object runs in the same OII; here, the system can be optimised by linking the OM and OII layers in a single address space). Running objects as separate processes provides memory isolation and

protection against bugs, however, this does not protect persistent state on disk from being tampered with. By jailing OII's, further protection can be provided, as jailing can guard access to (per-object) directories and communication endpoints. See examples below.

Some options for the MOS to run objects are shown in Fig. 31. These options are governed by MOS startup flags. As shown in Fig. 31, a MOS can be instructed to jail or group objects in different ways. The MOS can run OIIs, with or without jailing; it can jail individual objects (OIIs running only one object instance).

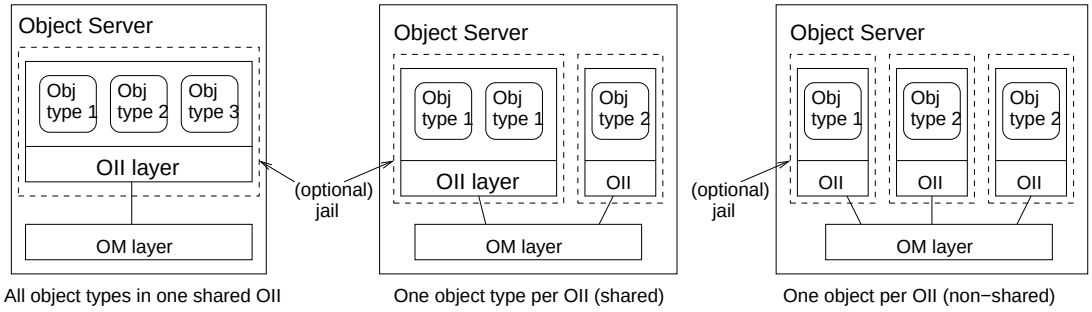

Fig. 31. Example configurations of a MOS running OII programs and objects

Conceivably, the MOS can be provided with a repository of ''trusted'' object types (e.g., the standard RMO implementation provided with Mansion), and that new object types care placed in an ''untrusted'' repository (e.g., world-specific object types, or possibly in the future objects provided by agents). A similar approach is taken in Java's sandboxing system. In the current implementation course-grained MOS startup flags govern aspects such as whether jailing is used for all OII's in the MOS at once.

The tradeoff to decide between above-mentioned options is typically whether or not the added security of jailing an object is worth the overhead (both in terms of increased latency for IPC and in terms of the extra jailer and OII processes needed).

Another scenario is where all objects are instantiated in the same OII process. If the implementation of these objects is trusted, the OII does not need to be jailed. In this case, the MOS can be optimised by including the OII layer directly in the OM layer, avoiding the overhead of IPC (and related memcopy operations) for shipping data across protection domains. Note that such optimisation has not yet been implemented; currently, the OII always runs as a stand-alone program in a separate address space in one of the ways indicated in Fig. 31, by default with one object per OII, unjailed. Note that if an OII runs in a jail, the jail must be configured so that the OII process can connect to the OM layer after startup. This connection is used to ship RPC requests over, in both directions (i.e., object invocations and replies, and runtime system calls from OII to OM).

7.5.4. Putting things together

Fig. 32 gives a schematic overview of how a binding to an object in a MOS is established. This schematics reiterates some of the points made so far. Shown are an object server (right) with three objects, each in an OII. To the left, a MMW process is shown that hosts an agent. The agent has a runtime system in its address space (the Mansion API), which contains a *bind* call. Binding is used to connect a stub (shown) to an object in the agent's room. As part of binding, the MMW creates an RPC connection to the appropriate object server (found by resolving the object handle in the location service), and it creates its own invocation forwarding endpoint to which the object stub can be connected. Shown is the situation where an object binding is established to Object 1 in the object server. The binding forwards incoming invocations to the appropriate invocation service of the MOS (right) in which the object server resides.

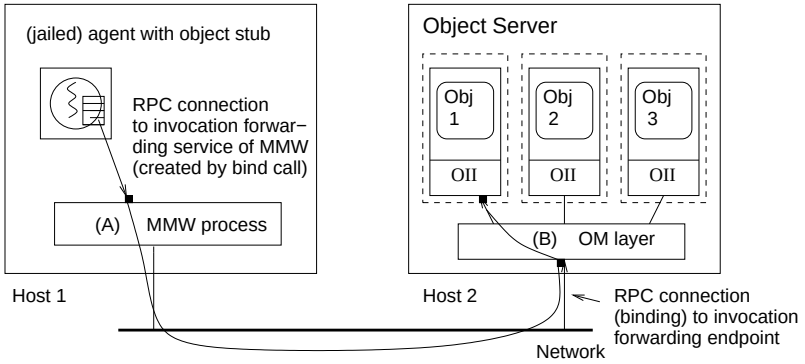

Fig. 32. Schematic of how an agent binding to an object is established. (A) and (B) indicate places where the marshalled ScID argument may be overwritten.

An invocation in Mansion takes the following route. The agent invokes a method on its object stub. This stub marshalls the invocation, and ships it to the invocation forwarding service of its MMW to which it was connected as part of binding. The MMW knows the AgentOwnerID of the agent, having looked it up in its internal agent table (chapter 9).

When an invocation from the agent is received, the MMW process replaces the first ScID field of the marshalled invocation by the agent's AgentOwnerID **(A)** (leaving the rest of the marshalled invocation unmodified), and then forwards the marshalled invocation to the MOS. The MOS receives the invocation and checks from where the invocation came. If the invocation came from a process within the zone, it simply forwards the invocation *as-is* to the OII in which the object resides. If the invocation came from a process outside the zone, then depending on configuration it either rejects the invocation, or it replaces the scid field of the marshalled request by the invoking process' ZoneID **(B)** (Sec. 8.2.10). Next, it forwards the marshalled request using an internal forwarding table to the OII in which the object resides, where the request is passed to the appropriate skeleton code for unmarshalling and invocation

of the object. The skeleton code will check the object's ACL before invoking the method, and returns an error if not allowed. Otherwise, it invokes appropriate object instance's method, and marshall the return value and *out* arguments. Return of the call takes the opposite route.

As a practical consideration, in many worlds, objects may have *default* entries to specify the permissions of arbitrary (not earlier known) agents or principals. Many rooms and objects in Mansion may contain public information. In such cases, an object's ACL is used to distinguish administrators from regular users, or to allow for creating exceptions for agents that can do more than others, or less—that is, to define more specific access control rules for specific agents (agent owners) in a world[43]. In other worlds, as indicated in Sec. 3.10, AgentOwnerIDs can correspond to a *role* or an identifier for a (world-wide) payment scheme, instead of an identity. This makes using ACLs more scalable. If useful, both approaches can be mixed; objects see just ScIDs for access control, irrespective of their semantics. This makes usage of the mechanism flexible.

The basic Mansion access control model is simple. ACLs are not a very scalable solution, but often effective, and if required world designers and object implementers can implement extensions to the model such as mapping ScIDs on roles, as sketched above.

7.6. Conclusion

This chapter describes the design and implementation of the Mansion Object Server (MOS) and the *MansionObject* access control scheme. The MOS is simple by design and secure. Objects can run alone or grouped per type in a separate process, possibly jailed to protect the system against potentially buggy or even malicious object implementations.

The Mansion zone model provides the basis for authenticating clients. If a client process runs in the same zone as the MOS, it is trusted. A MMW process can pass the AgentOwnerID of an agent it hosts to the MOS, which passes it to the object for access control. The standard *MansionObject* access control model is simple and straightforward and uses an ACL with a role bitmap per principal (with possibly a default role) for each object. If required, more sophisticated access control mechanisms can be implemented by objects internally using the *ScID* argument that is passed with every method invocation.

The MOS is a stand-alone program which is not dependent on other services, such as a location service. The client program which creates an object has to register the MCR of the object in the location service. A simple mechanism for active replication of objects is conceivable. (Note that in the case of replication, the MOS or a specific object in the MOS *may* depend on another service, such as a sequencer—see App. 4).

[43] In addition to using the standard ACLs provided by *MansionObject*, which function on a per-method basis, objects can also implement additional access control policies that are based directly on the ScID that is passed passed with each method information. If the ScID is a real agent owner ID or a pseudonym, the object can maintain (persistent) state with the invoking principal that can be recovered when the same agent(owner) invokes the object a subsequent time. Such approaches—and their side-effects, for example, privacy issues similar to those that occur when using cookies in the Web—are outside the scope of this dissertation.

The MOS uses Mansion RPC based on authenticated, ordered connections provided by the zone authenticated communication (ZAC) layer. The object IDL extends the Mansion RPC IDL, and allows for straightforward specification of object types and automatic generation of stubs and skeletons. Stubs can be generated for C and C++, but other languages are possible. Objects are currently implemented in C++. The Mansion RPC system internally uses a simple XDR-based marshalling format for portability.

.

.

Chapter 8
The Mansion middleware

Mansion is supported by a middleware system that hides operating system details from agents and presents an interface to it. The Mansion middleware (MMW) drives Mansion: it starts, manages and migrates agents, resolves location independent identifiers and creates bindings to distributed services and objects, establishes and manages interagent connections, and interacts with the RMO for local (room-internal) operations.

The interface provided to agents, the Mansion API, implements the logical model of Mansion. It mediates access of agents to objects and other agents according to the logical constraints imposed by Mansion.

A Mansion world is a distributed system consisting of multiple components on different machines in different zones. The MMW is the glue between these components. Its function, design and implementation is described in this chapter.

8.1. Introduction

Agents are provided with a runtime system that provides the Mansion Application Programming Interface, the *Mansion API*. The functionality of the Mansion API is implemented inside the Mansion middleware (MMW) process. The MMW's main task is to host and manage the life cycle of mobile agents, that is, starting, suspending, migrating, and stopping them, and implementing the calls provided by the Mansion API. The MMW, in turn, depends on a number of (distributed) services that are needed to make a world work. Examples are object servers, the location service and the basement.

The Mansion API is needed for agents to interact with Mansion entities in a controlled way. The MMW ensures that appropriate protection mechanisms are in place so that agents cannot escape the logical (security) constraints imposed by Mansion. As such, the MMW acts as a *reference monitor* for agents. A reference monitor checks whether invocations made

adhere to the system's (security) policy, and denies operations which do not correspond to this policy [11]. Besides checking security aspects (e.g., resource limits), the primary function of the MMW is to ensure that invocations adhere to the Mansion conceptual model, for example, that an agent may not bind to an object in another room.

The MMW has a modular structure. It has distinct components that deal with specific aspects of the system, such as mobility and communication. The main components and (protection) mechanisms of the middleware are described in this chapter. It also describes relevant aspects such as agent migration and the implementation of the resolver for accessing the location service.

8.2. The Mansion middleware: functional view and security

The main Mansion middleware (MMW) process manages agents, and mediates access of agents to a system and to the resources required to get work done: access rooms and the objects therein, and migrate. Sometimes they need to communicate with other agents to get their work done.

Agents require local resources, such as CPU time, memory and disk space (most programs write temporary files, for example). They need to access files such as libraries shipped along with the agent or libraries or programs provided by the local operating system[44]. Agents should not be able to use more resources than granted. The jailer is used for this; it has been designed to protect system resources against (potentially malicious) agents. Besides local resources, the system as a whole also needs protection against rogue agents; using suitable parameters checked and enforced by the MMW, global resource management protection can be achieved. This section describes how the MMW implements global and local agent life cycle management, agent shipment, binding to objects and interagent communication.

8.2.1. Agent life cycle and resource management

An important function of the MMW is to manage each agent's life cycle. In principle this amounts to receiving, starting, stopping and migrating an agent. However, resource usage must also be accounted and acted upon; for example, an agent that exceeds a limit of CPU-time for a period of time should be suspended or maybe even killed. Similarly, an agent that has exceeded a global *time to live* has to be killed and shipped to the morgue. Agent life cycle management entails two things:

[44] If agents in a world have common requirements, for example on availability of basic programs such as a perl interpreter, the world administrator can request all zones to install such programs when adding zone members to the world; similarly, a world designer may require availability of at least one machine with a standard operating system such as Linux per zone, to allow execution of binary agents. A negotiation protocol may be required for the MMW to negotiate transfer to such a host at zone entrance time (future work, see also [72]). This may require a *zone information system* and a *zone entrance daemon* per zone to replace the current round-robin load balancing scheme. These are not currently implemented.

1. Global (possibly continuous) management of global life cycle aspects of an agent, such as an agent's total lifetime in a world; global limits can be defined in the world design document or, at world entrance time, by the world entrance daemon.
2. Local life cycle management of each agent's process on the current machine. This involves controlling aspects such as starting, suspending, resuming, and killing the agent, for example if some local or global resource usage limit is exceeded.

Sometimes, global life cycle related aspects are translated to local resource limits, such as a maximum AC size or a time to live. Mechanisms also need be in place to protect global resources in view of sloppy or malicious (non-)enforcement by MMW systems; these are described below.

8.2.2. Global life cycle management

Global life cycle management consists of enforcing global constraints on an agent's resource usage during its lifetime in a world. Examples are the cumulative time that an agent and its children are in a world, which relates to the amount of CPU time used by an agent and its children, or the amount of network bandwidth and storage that a given agent uses. The latter is related to the (cumulative) maximum amount of data that is stored in the AC of a given agent and perhaps a maximum (#bytes) on interagent communication.

Controlling agent resource usage may imply defining and checking parameters such as the maximum total amount of data an that agent is allowed to store in its AC, and taking action when this limit is exceeded (possibly, returning an error to the agent when it tries to follow a hyperlink, or killing it and sending it to the morgue).

The world entrance zone is responsible for defining world limits at agent injection time; these are parameters used by the world entrance daemon and hardwired into the agent's AC (the initial AC is signed by the world entrance daemon). There may exist default limits or global maxima defined by the world owner, as well as possible ways to extend limits—for example, by payment.

Examples of global parameters that can be set at world entrance are:

- The number of *hops* (machines) that an agent is allowed to visit
- A limit on the number of child agents of an agent, and their hopcount limit
- A global time to live (cumulative or as an expiration time) for agent and children.

The above parameters help protect the system as a whole against rogue agents that use too many resources, erroneously or as the result of an attack. Some parameters may be translated to locally enforceable resource parameters per agent such as:

- The maximum AC size for an agent
- A maximum runtime per machine

In addition to per-agent limits, there may also be per-agent owner limits. An example is the total number of agents that may be created by an owner, the number of agents that may run at a single time, or the cumulative AC space or bandwidth or CPU time used by them. Such limits cannot be enforced decentrally in a secure way and need to be kept track of and checked by the WEZ. Global limits are typically set at agent (owner) registration or at injection time. Resource limits may be related to payment.

8.2.3. Enforcement and verification

Enforcing local resources is typically straightforward: an AC size limit can be enforced by checking the size on arrival using its table of content, and checking how many bytes are added by AC operations. CPU time can be accounted for using existing OS calls or by querying the jailer; in other cases, an enforcement of a local policy can be delegated to the jailer (e.g., maximum number of processes, memory, disk space)—but these resources are often governed by a local policy.

Global (world-wide) resource management poses a bigger challenge. This is due to the world owner or the world entrance zone not governing every agent's execution—this would not scale and would conflict with autonomy requirements for hosts and agents in the system. As a result, the world entrance zone cannot enforce limits while an agent is executing on a host in a zone. There are two moments where the world entrance zone can influence what an agent can do in a world. Both moments involve a component of the world entrance zone:

- Physical agent migration
- Cloning an agent

Below, central and decentral global resource checking mechanisms are described.

Decentral checks. When an agent migrates, its contact address needs to be updated on the agent location service. In Mansion, this action requires both the sending and the receiving middleware to agree on shipment; before the receiving middleware commits the update, it has received the AC so it can verify certain parameters. Decentral checks are possible on whether the agent's properties do not exceed the parameters defined in the AC, such as the total AC size. Exceeding a time to live can also be detected at the time an agent follows a hyperlink. If some limit is exceeded the protocol for updating the contact record in the ALS can be aborted.

The world depends on trust in the (majority of the) middleware processes for the above to work. Should processes cheat and (together) decide to accept agents which have exceeded the limit, the world entrance zone could conceivably detect this at best when the AC is shipped to the Morgue (the agent is extracted there, and the Morgue can use the agent's then-visible *audit trail* to check the time, location and size of all modifications to AC segments, see Sec. 8.3.1). Enforcement could become difficult if cheating becomes structural. This is however unlikely if the parameters are reasonable: the parameters work to protect the world as a whole, so it is likely that most zones would cooperate to enforce them. Typically, the above mechanism will stop an agent that has exceed its limit, if not immediately then at the next transfer.

The importance of decentralized verification is that it scales: the ALS is not involved in making checks, while the update protocol provides the basis to have middleware processes do so. If needed, the process can also provide the basis for auditing. If the world owner has imposed requirements when accepting a zone in its zone list, it can have the Morgue check at regular intervals whether the middleware processes in the world comply.

Central checks. Certain checks *can* be made centrally, always or periodically, if required. An agent's *hopcount* can be checked easily and cheaply in the ALS, as it is straightforward to maintain a counter for each AgentID. If an agent *clones*—a process which involves the world entrance daemon, Sec. 9.1.7), the WED can check a simple set of counters which hold information about the number of agents cloned by its parent, and if relevant and the parent of its parent (there may be a maximum depth), or a global per-agent owner counter, depending on policy. This prevents an agent or its owner flooding a world with agents.

Since the cloning protocol involves sending the AC of the parent agent, or the ToC thereof (a future optimisation, see appendix 6) to the WED, it can keep track of the cumulative AC size of all agents of an owner with some certainty[45] and enforce limits on this by refusing to clone (effectively, by refusing to create a new AgentID).

The above measures pose inescapable consequences for agents that do not adhere to limits. If an agent could migrate without updating its contact address—which can only occur with colluding middleware processes—it would become untraceable and could no longer be reached by other agents (or its owner) in a world. Normally, a MMW process will not accept an agent when its currently registered contact address does not match the zone that the agent is migrating from. So, even if an agent is allowed to migrate without updating its contact address in the ALS, it will not be able to go very far. If an agent exceeds a limit, which may be detected while it is running or at migration time, the MMW on which it currently runs can send the agent to the Morgue. If limits are exceeded before an agent migrates, for example the agent's time to live, the MMW that currently manages it can decide whether to kill the agent or give it a chance to exit gracefully[46].

[45] A threat could be that a malicious middleware conspires with an agent to clone an agent several times *before* they accumulate data in their AC, thus allowing many clones to be accepted which may later embody a huge cumulative AC size; a world owner would thus do well to carefully choose a maximum number of agents in combination with a maximum AC size per agent, to protect against this threat. A similar issue may hold for other parameters as not every detail can be checked centrally with certainty.

Global resource limits (such as an agent's time to live) are encoded in a special segment in an agent's AC which is created and signed by the WED at world entrance time. Agents can use a Mansion API call to inspect their resource limits and usage at any time (Sec. 9.1.2); the information returned by this call can be adapted to include relevant resource information that applies to a given system configuration or world.

There is no such thing as a global ''kill switch'' for agents. Due to Mansion's distributed design, this is impossible. MMW processes are autonomous in how they manage agents. Agent owners can request their agent(s) to exit the world, by sending a predefined message to them. An agent's parent—if the agents are so programmed—may be able to kill its children in a similar way. But agents are autonomous and decide themselves whether and how they respond to requests. Only the MMW that manages an agent can kill it, for example if it exceeded one of its limits; then *its* autonomy exceeds that of an agent. Of course a MMW that observes an agent gone haywire should take action.

An agent may also get killed or suspended because the owner of the machine on which the agent's MMW runs needs to use the machine. This may occur in open worlds that use a volunteer computing-like model to get resources for agents to run on. This is similar as the policy used in desktop grids: there are no guarantees that jobs will complete their task or even return. Suspension of agents is an option, subject to implementation. Suspension is tricky as it means that the agent's time to live may expire while it is suspended. ''Desktop liberation'' policies have not been studied in this thesis, but have been studied elsewhere (Sec. 1.4.4). Note that there is also no global mechanism for dealing with killed or unresponsive agents whose status is unknown[47]. Mansion currently takes a best-effort view, meaning agents may be killed and lost, although the MMW will attempt to ship its agents to the Morgue if it gets time to shut down gracefully.

8.2.4. Processes and local agent life cycle management

In Mansion, each agent is a process. A process consists of one or more threads in an address space. Different agents cannot access each other's address space. Process abstraction provides a simple and secure execution model for agents, which is portable across (UNIX) operating systems. A similar abstraction can be implemented in most other operating systems, including most versions of Microsoft Windows based on the NT kernel. Agents may spawn subprocesses; these are executed in the same jail as the original agent—every agent has its own jail.

[46] In UNIX, a special signal (e.g., SIGHUP) can be sent to the agent a few seconds before its time to live expires, to trigger it to write important information to its AC before it is killed; an alternative method to inform agents of (looming) limit exhaustion is to return a special error code next time the agent invokes an API method.

[47] There is no reliable mechanism to find out if an unresponsive agent is killed, suspended, or simply lost. The only information which can be recovered with some certainty, is the host that an agent was running on at the time it got lost, as this information is stored in the ALS. Recovery mechanisms must be stored at the application level by agents, if required.

In UNIX (and other operating systems), it is straightforward for a parent process to monitor its child processes. Actions such as *suspend, resume* and *kill* operate fast and effectively in most modern operating systems. The UNIX process abstraction is therefore an effective basis for local resource management.

Agents are started up, in a jail, by their Mansion middleware process. The MMW monitors all its agent directly. Since the MMW is the reference monitor that controls an agent and its invocations in the system, it is important that the MMW has direct control over the agent processes that it manages. It should be impossible for agents to bypass security mechanisms or logical constraints. The MMW should be able to kill an agent if it misbehaves, or suspend an agent temporarily if it consumes too many resources, or delegate this responsibility to the jailer (chapter 6 and appendix 3).

Mansion is designed to support agents written in different programming languages; the programming language or binary type (for a compiled agent) is annotated in the AC by the agent's owner. A special segment contains information on the agent type, possible options (e.g., when an agent is precompiled for multiple platforms) and dependencies (e.g., libraries for different systems if the agent is not statically compiled). The MMW selects one of these if it receives an agent[48]. If needed, the MMW can start up an agent in a suitable interpreter, such as the Java virtual machine (JVM) or a python interpreter. Note that Mansion does not trust the interpreter any more than it trusts the agent; it is viewed as part of the agent. Due to weak mobility no adjustments of interpreters are needed to support strong mobility.

Mansion supports multiprocess agents. A multiprocess agent can, for example, be a (binary) agent process that is started from a script which also spawns other processes for certain subtasks. Mansion groups all processes of an agent in a jail. The jailer is a trusted Mansion component and can be instructed to kill or suspend all agent processes or threads using a protocol, currently using UNIX signals.

8.2.5. Alternatives to process-based agent management

Many mobile agent systems—AgentScape, for example—implement life cycle management using **agent servers** [127]. Agent servers are language-specific execution environments. For example, a Java agent server can consist of a Java program, consisting of one Java thread which waits for incoming agents, and starts each of them as a separate Java thread. Here, one agent may influence execution of another agent.

Most languages—or operating systems, for that matter—are not designed with the notion of executing mutually distrusting threads within the same interpreter, that is, within the same address space, which makes it hard for agents to keep secrets from other threats or defend themselves against malicious agents executing in the same agent server. Few if any

[48] The agent code description segment may describe an ordering, that may be autonomously evaluated by the receiving middleware. For example, it could indicate a preference for binary agents with a fallback for an interpreted agent in case a zone has no system supporting the indicated binary type, or in case execution failed. These aspects were not implemented in the current system.

interpreted languages implement functionality such as starting/stopping threads within a single address space in a secure way [25, 14]. This makes agent servers that are based on a thread model a less suitable choice for systems that require a large degree of security between and controllability over potentially malicious agents.

One conceivable advantage of using agent servers is that these could minimise resource usage (e.g., memory footprint, CPU usage) by running different agents as threads in the same address space, compared to executing agents as separate processes. However, process execution is rather efficient on most current-day operating systems. For example, the memory overhead of executing multiple interpreters simultaneously is limited due to shared instruction pages. Five JVMs effectively use the same code space as one[49].

8.2.6. Agent containers

Agents carry state. They contain an agent's code and initialisation data, and often require files as input. An agent's code may be precompiled for different operating systems. The MMW may also need to associate some information with an agent when it migrates, including data received from currently open interagent connections which are not read yet. In addition, an agent may require other programs (possibly precompiled for different platforms), library files, input data, etc., Finally, the agent may pick up data and store results. The agent stores this data in its AC.

Internally, the MMW uses the AOS AC, with similar API calls to create segments and read and write data[50]. These can be simplified slightly, as the agent need no access to segment types and cookies. Agents can only access AC segments of type *DATA*. *SYSTEM* segments are not accessible to agents. These contain code for internal use of the middleware; this information can, if useful and if stored in a persistent segment, be viewed by the agent's owner and by MMW processes, but is little use to agents.

An example of a system segment is a segment similar in function to an *Agent Passport (AP)* An AP binds an agent to its owner for authentication purposes. the ''AP'' only binds an agent to certain properties such as its owner; it does not imply trustworthiness of the code. The agent-agent owner binding is implemented by the WED which signs the first content of the AC. It then includes a persistent segment containing the agent owner's *AgentOwnerID*. It also contains information about the Morgue to which the agent should be shipped at exit time, the agent's AgentID, resource limits, and possibly other information required by all MMW processes visited by the agent.

Segments may be persistent or transient. A *persistent* segment may not be removed from an agent's AC anywhere on the agent's following itinerary; the AP segment discussed above

[49] The same applies to a jailer, but here the offset is that a jailer imposes system call interception process switching overhead. This is considered acceptable for Mansion since security is considered more important than raw performance.

[50] The content of segments in an AC must be stored in a platform-independent way by agents when, for example, writing binary data to a file on a little endian machine which must be read in later on a big endian machine; the content of segments in an AC are opaque to AOS and MMW.

is persistent, as it is needed for authenticating core properties of an agent on each visited MMW. A *transient* segment may be removed at any time. The MMW processes can verify the integrity of the AC by checking that persistent segments have not been modified or removed after they were marked persistent. The AC is designed such that such integrity verification is efficient (Sec. 4.4.1, 8.3.2).

8.2.7. Starting an agent

Previous sections and chapters described the basic components needed for agent shipment and execution. Here is the procedure of how the MMW receives and starts an agent.

1. An agent comes in by means of the agent transfer protocol described in Sec. 8.3.2. The AC is checked: does this system support an agent programming language from the ones contained in the AC? Is any limit exceeded?
2. If this checks out, a new entry is created in the *agent table* of the MMW. A scratch directory is created by the MMW in which, by convention, a set of files (e.g., libraries) may be copied into from the AC. The selected agent binary (or interpreted program) is copied to the jailing directory. The jailing directory is registered in the agent table.
3. The jailer is passed a TCP port number allocated by the MMW, on which it waits for a binding connection from the agent; this is needed to enable the agent to instantiate its Mansion API stub library and object stubs and connect them to the middleware. This port is passed to the agent as a commandline argument. The agent is also provided with a key in its jailing directory with which it can authenticate to the Middleware. The agent table is updated accordingly.
4. The agent is started by the jailer; if agent startup yields no errors, the agent first initialises its runtime system, which includes binding to the Mansion API and invoking an init method on it.
5. If the the agent has called the Middleware by binding and invoking its runtime interface, the MMW believe it is running; it will it (re)instantiate the agent's communication buffers—if needed by reinstantiating suspended buffers from the AC (Sec. 8.2.13) and ensure the agent's communication endpoint is reachable to outside agents.
6. The MMW commits the AC migration on the agent location service and register the agent's communication endpoint there, before setting the agent's state to RUNNING.

The order of the above steps is important. The moment the AC migration is committed on the ALS and the agent's new communication endpoint is registered by its current MMW, its state is changed from *MIGRATING* to *RUNNING*. Next other agents may connect to the incoming agent communication interface of the MMW to connect to the agent or write data to incoming message buffers of already established interagent communication channels[51]

[51] The buffers have a maximum size, also dependent on the AC's maximum size because an agent should be able to migrate to

After all this is completed, a message is returned to the sending middleware, which will then kill the agent and remove the agent's AC using an AOS operation. Should the sending middleware—a malicious case—not remove the agent and let it run, it will simply be an unofficial copy of the agent, that is unreachable as the AgentID now maps to the next machine. Worst case, the sending middleware could eat a few incoming messages from agents that were not yet aware that the agent had moved; by checking the ALS for the agent periodically, these agents (or rather, their middleware) should eventually notice.

Each agent in Mansion is started up inside a jail. Each jail has its own private directory in which an agent can write files, the *jailing directory;* this directory contains a code directory and a directory for data files. The MMW accepts a script as the initial program to start. System files or directories (like, */bin/*) are typically accessible read-only. The jailer manages and controls all threads and processes in the jail, including child processes created by an agent, and allows a network connections to the MMW to invoke Mansion API calls. The MMW passes parameters to the jailer for basic resource management as described earlier, and can use signals to suspend, restart or kill an agent and—once the prisoner is killed—its jail.

8.2.8. Runtime system

Agents interact with the middleware (MMW and, indirectly, object servers) using stubs of the Mansion API and object interfaces. Stubs are simple libraries that can marshall and ship methods and their arguments. These stubs provide the *language binding* for the agent. All interfaces in Mansion, including the API, are specified using the Mansion **interface definition language (IDL).** Mansion comes with an IDL compiler, which currently generates stubs and skeletons for C and C++ (Sec. 5.2.3). Because the internal data representation is simple, it should be straightforward to generate stubs for, for example, Java, Python, and other languages.

The Mansion API is implemented in the MMW; the stubs themselves implement no functionality. The reason for taking this approach is twofold. The first is simplicity, as porting a runtime interface stub is straightforward. The second is that this way, agents cannot tamper with the MMW implementation. The stubs are linked in the agent's address space, and are initialised to connect to the MMW process which started it. Over this connection, invocations are shipped to the MMW, and replies are returned as simple messages consisting of basic types available in any programming language, such as integers and strings.

The Mansion API provides basic methods to an agent in a world. Mansion provides a small library (written in C) which wraps around the Mansion API to provide convenient methods for common tasks, such as storing files in an AC. This library is comparable to the way in which libc wraps around raw UNIX system calls. In addition, files stored in a specific directory in the prisoner's private jailing directory are automatically saved / restored from the

another host without having read the incoming data, and in that case there must be room to store the messages. Obviously, there is some room for lenience but not much, also to prevent too many local resources from being used.

AC at migration time. This simplifies implementation of agents somewhat, although similar functionality can be implemented manually by writing information to data segments in the AC.

8.2.9. Agent authentication and access control

Authorisation in Mansion is based on **access control lists (ACLs)** associated with objects. Every Mansion object has an ACL. The ACL of a room monitor object governs access to a room. It is consulted when an agent comes in, before an agent is allowed into a room; if an agent is not allowed in by the RMO's (default) ACL, migration fails.

Objects in Mansion are invoked by the MMW on which an agent runs. The MMW passes information an agent owner's AgentOwnerID (obtained from the AC) to an object at invocation time. Based on this information, the object can determine whether access to a given method is allowed, as explained in chapter 7.

The AgentOwnerID currently corresponds to the agent owner's public key. In very large-scale worlds, authorisation based on agent owner information may not be feasible; although effective and simple, ACLs do not scale well since every object has to maintain a subject-permissions mapping. This makes that ACLs can soon grow extremely large. In practice, a default entry in an object's ACL solves this problem (possibly allowing additional entries to *block* access, rather than allow it). However, if meaningful access control is to be applied, it is likely that a more scalable mechanism needs to be applied.

The Mansion ACL system can be used to implement attribute or role-based authorisation. For example, when a world supports a payment scheme, the semantics of the ScID that is passed to objects[52] may correspond to a payment scheme identifier. Similarly, it may correspond to a role. To make this concrete: a world may have a set of roles, each indicated using a 32-byte random string that has the same size as a ScID. The size of a ScID is sufficient to encode a huge set of roles or attributes. The world entrance daemon associates the appropriate role to the agent by storing it in the AC. This ScID is then passed to each objects that the agent invokes. In addition to this role ScID, each agent also has a unique AgentID; this allows the world entrance zone services to easily (re-)identify the agent owner, if needed.

Like all application semantics, the object should be programmed to understand about roles. Information about the roles of a world and their semantics should be passed to object programmers and content providers using external documentation.

Besides having an ACL for basic per-method (yes/no) authorisation, the ScID field is passed with each object invocation. Object programmers may thus modify the object's behaviour depending on the invoking role or attribute. As far as mechanism is concerned, this is identical to how access control is applied when using AgentOwnerIDs.

[52] This ScID is simply placed in an AC SYSTEM segment; by default it corresponds to the AgentOwnerID but to the middleware and the object server, the semantics is opaque.

Summarising: ScIDs used for authentication in Mansion may correspond to an AgentOwnerID, a pseudonym, a role, or a payment scheme. Object ACLs can contain a ‘‘default’’ entry. If an agent’s AgentOwnerID is not in the ACL and a default entry exists, the default permissions are granted. With role-based ScIDs, a default ACL entry should not be created, since a default role should then be assigned at world entrance. Without a default ACL entry and if an invoking agent’s ScID is not in the ACL, access is denied.

8.2.10. Mansion object server modes

The Mansion object server (MOS) was described in chapter 7. For clarity, it is however useful to describe the two modes in which it can operate. A MOS can run in two modes:

- A mode for running regular objects, which are objects that can be invoked by agents in the zone. In this mode, the MOS can only be invoked by middleware processes within its own zone, which pass the agent’s ScID with each invocation. This ScID is passed to the object.
- A mode for running ‘‘service objects’’ (or ‘‘services’’) that are not invoked by agents but can be invoked by middleware processes anywhere in the world. An example service is the zone location service. In this case, the the MOS passes the ZoneID of the client—as authenticated by the MOS—to the service object to check its access control list.

The interface for creating or deleting objects can only be invoked by a client process in the same zone as the MOS. Depending on a configuration option, it has to have exactly the same zone member key as the MOS (Sec. 3.8.3); this means that only the (local) user that instantiated the MOS process[53] can create objects in it.

The object server is a critical component in each zone. There must be at least one, but typically there are two object servers in a zone: one running in service mode to host the zone location service, and another to host regular objects including RMOs. Room monitor objects or other objects may be created in any MOS in the zone, on any machine. Depending on configuration, objects can be jailed, to protect the system from potential bugs or malicious objects. It is also possible to jail only objects of specific types (e.g., of a given C++ class).

8.2.11. Binding to objects

While the Mansion API is a static run-time interface, object interfaces may be loaded dynamically when needed. Every room can contain one or more objects (including the RMO), to

[53] Currently, objects in a zone are created manually using a set of scripts. These scripts create an object server, and then create an object in it. The object server itself is simply a program run from the command line. It is straightforward to create different object servers on different machines. Passing a valid zone member key to the MOS when starting it up makes the MOS a member of the zone. By default objects are not jailed, but passing an argument to a script is sufficient to do so.

which agents can bind. A **binding** consists of creating an object's interface in an agent's address space, and connecting it to the object (Sec. 2.3.10). How an object is instantiated depends on the programming language and runtime environment of the agent. In case of C agents, all object interfaces of a world are currently statically linked to an agent. In case of a C++ or Java agent, the interface may be instantiated from a class object [109] which is either compiled in or available from a statically or dynamically linked library.

An object's interface in Mansion is implemented as a stub interface that communicates with a remote instantiation of the object using an RPC mechanism. Connection setup is mediated by the MMW, and RPC calls are routed through the MMW. Setting up a connection so route RPC requests to is called binding.

The Mansion API contains a *bind* call which takes an identifier of an object (relative to the agent's current room) as an argument. Upon receiving this call, the MMW verifies if a binding to a given object is allowed using the object's ACL by calling the object server that contains the object. If binding is allowed, the MMW establishes an RPC forwarding service endpoint to which the agent can connect. Next, invocations on the object interface are routed to the appropriate remote object instance in a MOS (Fig. 33).

Fig. 33 shows two agents and three objects in a single zone. Both agents are in the same room; they are connected to the same RMO, located in the object server on host 2 (solid line). Both agents have bindings to objects in different object servers.

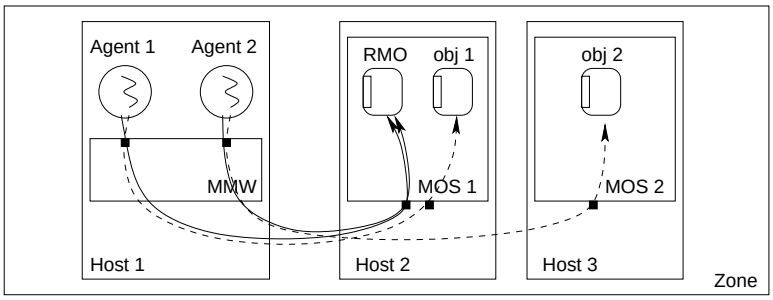

Fig. 33. Example bindings and internal connection routing through the MMW. Solid lines are bindings to an RMO; dashed arrows bindings from agent 1 to object 1 and agent 2 to object 2. Square dots are communication endpoints.

The small black boxes indicate communication endpoints. The agent's stubs connect to an endpoint on the MMW; the MMW routes RPC calls to an endpoint of the appropriate MOS. Note that the figure is a simplification: multiple communication or object endpoints may be multiplexed over a single TCP endpoint (port), so different endpoints do not necessarily use a different TCP connection, and different connections between the same MMW and MOS may be multiplexed over a single TCP connection as an optimisation. More details on endpoints were given in Sec. 5.1.

8.2.12. Implementation of the RTS bind call

The Mansion API runtime system exposes an internal *bind* call to agents. This is used to allow an agent to connect a stub to an appropriate RPC forwarding port, that is, to connect it to an object. The *bind* call returns an MCR. This returns the endpoint of an invocation forwarding RPC service provided by the MMW. This invocation forwarding endpoint is created by the MMW so it can forward an agent's marshalled object invocations to the appropriate Mansion Object Server (MOS), whose contact adress is looked up first, and an internal binding established by the MMW so invocations can be forwarded to it.

Marshalled XDR data is forwarded as RPC payload data. Mansion object method invocations are also marshalled using XDR; the compiler used to create object stubs (in C or C++) makes use of the Mansion XDR compiler.

For object invocations, all marshalled invocations are prepended by a header. This header is part of the *invoke* call of the MOS RPC interface. The header contains an object identifier. Using this identifier, the MOS can locate the appropriate object to invoke. The object identifier corresponds to the *index* field in the MCR of the object; this MCR is registered in the Mansion location service. Object invocations are thus layered on top of regular RPC invocations as nested RPC calls.

8.2.13. Interagent communication

Mansion provides calls for agents to communicate with each other. In contrast to most mobile agent systems, which typically provide a message abstraction [52, 126,19], or allow method invocations on another agent's (proxy) interface [58], Mansion can set up connection-oriented, reliable, ordered and secure connections between agents.

The Mansion API provides agents a socket-like interface to communicate with each other, without having to worry about aspects related to mobility, such as (temporary) disconnects: connections remain in place even if agents migrate physically. This section describes how migration transparent channels are implemented.

The Mansion API provides the following BSD-socket-like calls: *connect, send, recv, close,* and *select*. If required, agent programmers can map an (ordered) message abstraction upon the Mansion channel abstraction, for example in a library for exchanging FIPA agent communication language (ACL) messages between agents [7].

Internally, interagent connections are layered upon secure connections between MMW processes. Each agent has an endpoint, created by the MMW. This endpoint is registered in the agent location service (ALS) as the agent's contact address. It is used by another MMW to establish a connection to that agent.

Because agents are mobile, interagent communication has a slightly peculiar implementation and semantics. Whereas in TCP socket communication, a successful *send* call indicates that the data is stored in a sender-side buffer in the local operating system, a successful

send call in Mansion indicates that the data is stored in a receiver-side buffer, that is, a buffer in the MMW at the receiving side (Fig. 34). *Send* blocks until the data is acknowledged by the receiving side or an error is reported back (e.g., if the target agent migrated or the underlying connection failed). If the receiving agent migrates after receipt of the data (in the buffer) but before reading it, the received data is transported as part of the agent's AC to the next MMW, which reinstates the buffer before receiving new data. Each connection has its own receive buffer at the receiving end from which the receiving agent reads in-order.

The use of a receiver-side buffer avoids the situation that, in case the receiving agent migrates, the MMW on the sending side will have to poll the ALS repeatedly after a successful *send* transferred data to the MMW's buffer, to see if the receiving agent is ready. Consider also what happens if a sending agent moves after data is sent in the sender-side buffer before the data is sent out, and the sending MMW is partitioned off the network or crashes. In this case, maintaining ordering guarantees becomes difficult. Either way, a solution with a sender-side buffer is more complex than using a receiver side buffer.

A simplifying semantics is that if an agent sends data data is while a receiving agent is migrating, and thus while no receive buffer is ready, the sending agent will be informed through a (transient) error on the send call. The sender will then have to retry later. This makes the interagent communication abstraction reliable and ordered, but not fully migration transparent. Fig. 34 illustrates the approach.

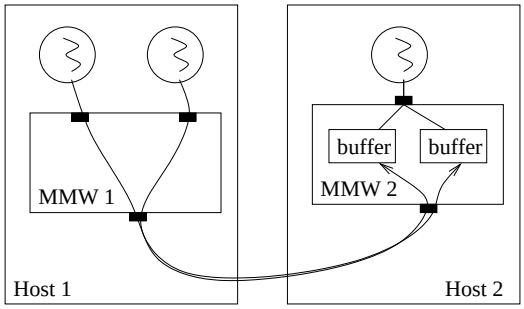

Fig. 34. Interagent connections. The flow of data over a channel in one direction is shown (arrows); the reverse route is identical to the one shown (but then with buffers in MMW 1). The relevant buffer for sending data is identified by a communication endpoint combined with a connection identifier. The connection identifier is created as the result of a connect call, and is persistent over the lifetime of the connection; the transient communication endpoint is created by the MMW.

Fig. 34 shows two interagent connections between different agents. Each agent has an RPC endpoint, to which connections can be made. This endpoint is registered in the ALS; after connecting, a *ConnectionID* is assigned, which is persistent. After migration, a connection can be reestablished using the agent's current MMW endpoint, as found in the ALS, combined with the ConnectionID. The connection identifier is used to (de)multiplex connections to the same agent.

Internally, the operation used to send data to the receiving buffer is an RPC call over a secure (SSL) connection layered upon TCP. This call takes the connection identifier, and results in data being placed in the receive buffer of the target agent. The target agent can subsequently read the data from the receive buffer. The socket-like interface for interagent communication provided by the Mansion API is currently layered over this simple RPC interface. The internal RPC service used to establish connections and to place data in the receiver-side buffer is called *SimpleComm.*

8.2.14. The SimpleComm communication service

SimpleComm is an RPC service which can be reached by external MMW processes and which implements the protocol for interagent communication described in Sec. 8.2.13. One SimpleComm method is used to create a connection to an agent. The signature of the SimpleComm connect call is as follows:

```
int connect(in: mmw_mcr_t connecting_agent_mcr,
            in: mmw_gaid_t connecting_agent_gaid,
            in: int connecting_agent_connID,
            in: mmw_gaid_t target_agent_gaid,
            in: int options);
```

This call is invoked by a client MMW process as the result of an agent's *connect* call. *target_agent_gaid* specifies the agent to be connecting to; locally, on the connecting side, a *connection_id* was already created which is passed to the receiving MMW. Along with the *connecting_agent_mcr* or the *connecting_agent_gaid*, this is stored in the MMW receiving the connection, so that it can establish the route back. The agent's *accept* call is merely a way for the connected-to agent to obtain the *connection_id*. If the agent does not want to receive the connection, it should explicitly *close* the connection.

SimpleComm's *connect* call is used both for room-local connects that take an EntityID of an agent and which are transient (close when one of the agent migrates to another room), as well as for global, migration-persistent connections. The type of connection is indicated to the recipient using the *options* field. If *option* indicates a transient connection, the connecting agent's *AgentID* (gaid) is not passed to the recipient. Note that this means that the connected-to agent or its owner can never find out what the AgentID of the connecting agent was – that is, the connecting agent is anonymous. If agents wish to exchange more information, such as their global *AgentID*s for longer communication, they should exchange this information explicitly at the application layer.

The *connecting_agent_mcr* field contains the Mansion contact record of the connecting agent MMW's SimpleComm RPC interface. This is an optimisation. Together with the *connecting_agent_connID* field, this allows the connected-to MMW to route back return data to the connecting agent instantly, without having to resolve the connecting_agent_gaid on the

agent location service first. Note however, that for room-local connections (where connecting_agent_gaid is zeroed), this optimisation is needed as it is the only conceivable way to exchange the required information for setting up a bidirectional channel when the connecting agent's *AgentID* is not passed.

As a second note, a connecting agent that uses an *AgentID* to connect to, but wishes to remain anonymous, can set a flag using the MansionAPI *connect_gaid* call to indicate it does not want to pass its gaid to its peer agent; in that case, the semantics are identical to the room-local connect case: the connection is transient and closed as soon as one of the agent migrates. In all cases, the connected-to agent can recognise whether a connection is transient/anonymous or not by inspecting the *gaid* out argument of the *accept* call; if this argument is zeroed, the connection is transient and the connecting agent is anonymous. If gaid is filled in (non-zero), the connection is persistent and remains alive until the connection is closed or one of the agents is sent to the Morgue.

If the connected-to agent exists and its status is *RUNNING*, a connection identifier is created which is returned to the connected-to agent as well as registered in the corresponding agent's agent table; the local connection identifier is returned to the caller. If the status of the connected-to agent is not *RUNNING*, an error is returned to the connecting agent. If the error is *MIGRATING*, the connecting agent should attempt to connect again later. Note that *send* operations on persistent channels may return an error code at the agent level: *ERROR_API_COMM_AGENT_MIGRATING*; this is a transient error and indicates that the agent should retry sending sometime later, when the migrating agent is (hopefully) finished migrating and a new SimpleComm endpoint (at the MMW where the agent now resides) is registered in the ALS. The mechanics of resolving the new SimpleComm endpoint of an agent is transparent to the agent and part of the implementation of the MansionAPI *send* call.

The SimpleComm *send_packet* method takes a connectionID and a character array and length as arguments. The data is then stored in the corresponding receive ring buffer of the receiving agent's communication endpoint, up to a maximum determined by the receiving MMW, possibly depending on available ring buffer space. Like socket write, the *send_packet* method's return value may indicate an error or the number of bytes actually sent, which may be less than *length*. Any read data is read as a whole, sequentially from the beginning of the data buffer; if an error occurs (e.g., the receipt buffer is evicted when the agent migrates), this is acknowledged to the sender; the *send_packet* call returns with an error. The amount of received data is indicated as a return value. Thus, receipt of ''packets'' is reliable – if acknowledged, the data is queued in the *inbuffer* of the receiving agent – and if that agent migrates, the sender is ensured that this queued data is packed in the migrating agent's AC except perhaps in case of a fatal error that causes the peer agent to be migrated to the Morgue.

If an agent blocks on a *read* or *select* call for a given communication endpoint, receipt of a new block of data raises a flag that unblocks the corresponding call. In the current implementation, this flag is implemented as a pthread condition variable that unblock the RPC method that blocked, which will next unblock. It can be useful to keep the size of packets relatively small in order to keep latencies low, since packets will only be acknowledged and

delivered as a whole. A maximum packet size can be defined world-wide for all MMW processes. A maximum is currently hardwired in the MMW implementation in a global header file, but a placeholder for this parameter is present in a special configuration section of the world design document (see Sec. 3.8.1).

The SimpleComm interface finally contains a *ping_alive* and a *close* call, which poll the agent's status, and close a channel, respectively. *ping_alive* is currently used in a loop in the MMW which regularly polls the status of an agent's connections; this is useful to preempt unsuccessful attempts to send data to an agent which has already migrated. A polling loop is triggered every second. For agents which are actively communicating with other agents, error conditions (e.g., if a target agent starts migrating, or if the target agent is being morgueified, Sec. 3.3.2) are returned via the *send_packet* call. The return value for errors is identical to what would be returned by an unsuccessful *send_packet* call.

Note that *ping_alive* or *send* may may return an error that indicates an agent is being shipped to the Morgue; although this status is indicated on the ALS too, and although the agent table entry of the agent that is being morgueified will be removed as soon as morgueifying has completed (resulting in an *AGT_UNKNOWN* error), reporting this status may be useful for a MMW to mark the connection as closed for when its agent uses a *read, send* or *select* call next. The polling loop indicated above runs every second, which makes it likely that migration and morgueifying events (which certainly take a couple of seconds to complete) are noted before the agent's entry is removed from its current MMW process.

For morgueified agents, the polling MMW need not enquire the ALS about the agent's status. For Migrating agents, the client MMW may regularly poll the old MMW for the agent's status, and query the ALS for the new agent's address only after an *AGT_MIGRATING* status has been replaced by an *AGT_UNKNOWN* status, indicating that migration was successful. (Should migration fail, the old communication endpoint will be re-instantiated, resulting in an *AGT_READY* status).

Polling the peer MMW processes is useful to prevent the ALS from being flooded by unnecessary status requests from many client processes. Depending on the status returned, the sending MMW may return an error to the sending agent, causing it to reconnect on a transient failure (as the result of the peer agent migrating), or to take note that the connection dropped permanently as the result of the peer agent closing the connection or exiting.

8.2.15. Implementation of confinement

From the middleware implementation perspective, implementing confined rooms is straightforward. If an agent enters a confined room, the MMW notices this before it starts up the agent in the room, by checking the parameter *ISCONFINED* in the RMO that the agent is migrating to. If *ISCONFINED* is ''*yes*,'' the agent will be confined. In this case, the MMW sets a ''confined'' bit in the agent table.

Setting the confined bit ensures that the Mansion API call *connect_gaid* that agents use to set up connections to other agents is disabled. The *SimpleComm* interface is suspended, so inbound connections are also refused, and existing connections are not re-instantiated when the agent is in the confined room. Combined with the properties of the jailer, this ensures that the agent cannot communicate with agents outside the room. In addition, the confined bit ensures that the agent cannot invoke methods related to its AC. Files created in the agent's jailing directory are also not written to the agent's AC.

The conceptual model of Mansion describes that agents must connect to the **guardian agent (GA)** in the room to export (filtered) information to outside the room, before it leaves the room (Sec. 2.3.11). The agent (or its owner) should contact the GA after it leaves the room, to obtain the information (if allowed, possibly in return for payment). This approach is conceptually clean, but somewhat involved because it requires a specific communication protocol at the agent level[54]. The current implementation simplifies this approach. Our implementation avoids having to implement a specific GA and interagent communication protocol specifically for confinement.

To export information from a confined room, an agent simply writes information into a file called ''export'' in its jailing directory. This export file contains the equivalent of what would have to be passed to the GA, for example, a list of files selected from a *MultiFileContainer* object in the room matching some query implemented by the confined agent. The precise format should be specified by the world designer, but it currently is simply a newline-separated list of file names. When the agent leaves, the export file is picked up by the MMW and stored in a special directory where it can be inspected by the room's owner as needed, for example, after receiving a request by the agent's owner. It can conceivably also be (encrypted and) mailed to the owner of the confined room, along with the AgentOwnerID of the agent which created the file, however this is not currently implemented.

In all cases, the GA stores some data into the agent's AC before finalizing it and shipping it out of the confined room; currently, this file is hardwired in the middleware but conceivably it could be obtained from a FileContainer object in a room, or be generated specifically by a GA program. Using this information, the agent or its owner can contact the room owner later, to obtain the data or negotiate terms. The information can also contain an URL through which the data can be collected by the agent's owner, potentially after payment, e.g., in an e-commerce room where an agent selected 3 music files. Applications of confinement will be discussed in chapters 10 and 11.

8.3. Agent migration

The final important task of the MMW, is to handle migration. The functionality offered to

[54] Note that agents should also be aware of whether they are entering or left a confined room; this is complicated by the fact that nothing is stored in the AC when the agent is in the confined room. A simple solution is to have the agent make a note in its AC indicated that it enters an AC. If after restart it sees that it is not in a confined room (by checking the *ISCONFINED* parameter of its current room), it apparently left the confined room again.

agents for migration is simple. Mansion uses a weak migration model, so agents are restarted every time that they follow a hyperlink, even if the hyperlink points to a room within the same zone or on the same machine.

After invoking a call to follow a hyperlink, the invoking agent is suspended. The target room's MMW is looked up and the agent's AC is *finalized* and shipped to the target MMW. There, various aspects are checked, such as whether the target room accepts the agent (ACL checking), whether or not any global resource limits are exceeded, and whether the agent's programming language is supported. If all this checks out and the target MMW accepts the agent and is able to start it, the agent's contact address is updated in the ALS, after which the agent process at the originating host is killed, and all its internal MMW state (including bindings) is cleaned up. If migration is not successful, the agent is resumed where it left off, with an error code returning from the *follow_hyperlink* call. A MMW process is autonomous and free to accept or deny the agent on any ground.

Mansion coordinates the agent transfer protocol at the middleware level to ensure end-to-end security, to allow for verification of an AC, and to construct an **audit trail** that allows for detection of illegitimate alterations to an agent container by malicious hosts on the agent's itinerary. Below sections describe the mechanism and performance measurements.

8.3.1. Middleware-level audit trails

AOS provides basic integrity protection for ACs. It can verify whether an AC's content corresponds to the ToC with which it was shipped. However, AOS cannot and does not know what was supposed to be in an AC in the first place: AC transfer and integrity verification involve only a single transfer of an AC. AOS is also not able to infer what changes have been made to an AC prior to the previous AOS kernel (from now on also referred to as a ''hop'') on which the agent ran. These aspects require application-level (or rather, middleware-level) knowledge.

Because AOS cannot know the content or migration semantics of an AC, AOS cannot make an informed decision on whether any malicious modifications might have been made along the the migration path that an AC has followed. Also, AOS does not support a particular PKI, and by design, it does not come with a specific (agent) location system; all these aspects have to be managed by the middleware. Middleware-level audit trails can facilitate verification of AC integrity over a multihop itinerary.

Establishing audit trails for mobile agents was first described in [55] for the Ajanta system. In this system, *append-only* containers are used for agents to store data in, and a specific audit trail mechanism is used to detect tampering with the append-only container. Compared to the Ajanta system, AOS agent containers are more flexible, as both persistent and transient files can be stored in the same AC. Also, the AOS AC is platform-independent whereas the Ajanta solution is Java specific. The AOS ToC is specifically designed such that audit trail construction and verification can be done efficiently.

In Mansion, an audit trail is established by storing the ToC of an incoming AC, together with the signature over this ToC created by the MMW process that shipped it and the public key of the signer, in a segment in the AC before it is finalized and shipped to the next host.[55] By retaining the ToCs of all hops an agent has visited as an (iterative) part of the AC, an audit trail is established. Using this audit trail, all changes made to an agent since its creation can be traced. Public key cryptography (signing) at all hops is used to iteratively sign the stages of the audit trail.

Because of its design, it is straightforward to check for changes to the AC by comparing the ToCs in the audit trail, using a binary comparison algorithm that iterates over all entries in the ToC from first to last. Mansion's audit trail mechanism was first described in [79].

In Mansion, audit trails are verified at each hop before an AC is accepted for receipt and an ALS update committed. An AC that was tampered with can be refused and thus *contained* on the MMW at which the illegitimate change was made. This prevents tampered-with agents from migrating and spreading through the system.

To prevent detection of deletion of part of an audit trail (''rollback'') when cycles in the agent's itinerary occur—for example, when an agent visits a (malicious) host twice, a log of (ScIDs of) all hops that an agent has visited is stored by the ALS as an independent audit trail that can be compared with the audit trail in an AC when it returns to its owner, as a means to detect rollback. The ALS thus plays a crucial role in securing Mansion: because both sending and receiving middleware must agree on updating an agent's contact record, containment of tampered-with agents on the current host becomes possible.

The next sections describe the agent transfer protocol at a high level and presents measurements of end-to-end performance of the agent transfer protocol, including audit trail verification, over a sequence of hops with AC's of increasing size. The discussion below omits a description of functional verification of agent properties—i.e., agent code support, global limits, and the like—as these were discussed before. This checking is an intrinsic part of the agent transfer protocol, but does not add to understanding performance of the ATP.

8.3.2. Overview of the Mansion agent transfer protocol

This section gives a detailed overview of the Mansion **agent transfer protocol (ATP)** constructed on top of AOS. The Mansion ATP combines the audit trail verification mechanism and the transaction-based ALS update mechanism explained in the previous section. Agents are only migrated officially by means of an ALS update—thus, an agent's contact information in the ALS indicates the agent's *official* whereabouts at a particular time—if both the

[55] Because the ToC must be available before it can be signed, the signature cannot be included in the ToC/AC; therefore, the signature is not officially part of the AC. Instead, the ToC signature is shipped separately (over an authenticated communication channel) to the receiving middleware, who stores it in the AC together with the ToC as it came in, after verifying the signature. The receiving middleware does not have to verify the integrity of the incoming AC as corresponding to the ToC; this is done by AOS. So, ToC generation and integrity verification takes place at the AOS level, ToC signature exchange and verification, as well as audit trail construction, take place at the middleware level.

sending and receiving MMW agree on an agent's integrity. Integrity verification includes a combination of (current) AOS-level AC integrity verification and middleware-level audit trail verification.

The MMW waits at an ATP endpoint for incoming requests. The ATP endpoint is a regular communication endpoint, to which other MMW processes can connect. The MMW determines if it accepts an AC based on authentication of the peer process, and on information embedded in an initial ATP request message.

The ATP protocol is outlined in Fig. 35. The AC transfer and audit trail verification protocol is explained in detail below. Performance measurements are given in Sec. 8.4.

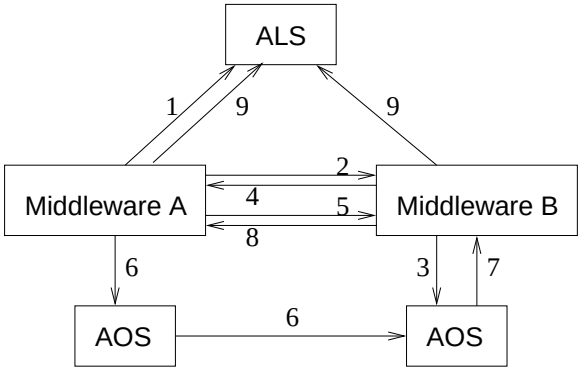

Fig. 35. Implementation of the Mansion hand-off protocol using AOS.

The Mansion ATP, including ALS update protocol and audit trail verification protocol, consists of the following steps.

- When a decision is made (typically, by an agent) to migrate an agent, MMW A suspends the agent and all its communication queues, finalizes the agent's AC, and initiates the ALS update transaction. As part of this, the MMW registers the intended target MMW's ScID in the ALS **(1)**.
- Next, a connection is made to the ATP endpoint of MMW B; MMW A provides information (e.g., programming language, resource requirements) about the agent to the target MMW through an *init* message **(2)**. Based on the init message, the target MMW decides if it is willing to receive the agent.
- If it is willing to receive the agent, it calls a method on the AOS kernel which creates an endpoint in AOS to which the agent can be sent **(3)**. A unique ''transaction identifier'' (XID) is also created by AOS, that is to be used by the sending MMW to ship the AC; this XID prevents that anyone can send arbitrary ACs to an AOS, and enforces that a middleware-level decision precedes shipment of an AC. The XID, along with the AOS ATP endpoint information, is sent to the client **(4)**.

- The sending MMW signs the ToC of the finalized AC using its own key. The ToC of the finalized AC is obtained by reading segment 0 of the AC using an AOS method. MMW A sends the signature over the ToC and its public key to MMW B for verification **(5)**, while simultaneously instructing its AOS kernel to ship the AC **(6)**.
- MMW B invokes a *wait_ac* call on AOS. This call returns an identifier for the AC after the AC is received and verified correctly **(7)**. An error is returned if there is any problem with the AC. After receiving a correct AC, its ToC can be read out by MMW B by reading segment 0 of the agent's AC. MMW B can now verify that the signature it has received from MMW A has been made over this AC's ToC.
- If all of this checks out, the MMW searches the AC (using a naming convention) for segments containing earlier ToCs. These segments are numbered sequentially, and the MMW can compare the signed ToCs using the audit trail verification procedure outlined before.
- If audit trail verification does not fail, the MMW checks functional aspects of the incoming agent to decide if it accept it; examples are programming language and global resource limits. If the MMW accepts the agent, the receiving MMW signs the ToC, and sends the signature back to the sending MMW as a receipt **(8)**. MMW A only commits the migration after it receives the receipt from MMW B; MMW B will commit the migration after establishing the agent's communication endpoint. Once both parties commit (or possibly abort, in case of problems) the ALS update transaction **(9)** is the migration transaction complete.
- Before the agent starts, MMW B stores the ToC, key, and signature over the ToC created by MMW A as persistent segments in the AC. This establishes the next component of the agent's audit trail. Once the agent migrates onwards will the MMW sign its copy of the AC/ToC, including the previous hop's signed ToC segment.

The steps outlined above ensure that agent integrity is verified at all migration steps, and that each MMW that sends an agent to another MMW process signs the agent's AC using its private key. In addition, AC integrity verification, ToC signature verification and storage of this information in the AC are required parts of the ALS update protocol. The ALS requires a commit by both sending and receiving parties. This enforces that both MMW processes sign the AC's ToC, ensuring that a verifiable receipt is sent and that a valid ToC is stored in the agent's audit trail.

8.4. Performance of the Mansion ATP

In Sec. 4.7, performance and scalability of concurrently shipping ACs was shown for the two implementations of the AOS kernel, one in Java and one in C++. The measurements

presented in this section focus on *end-to-end* performance of middleware-level agent transport, with a breakdown of component cost such as multihop audit trail verification and base AOS performance of the C++ AOS kernel used by Mansion for various AC sizes.

All tests are conducted on a dedicated cluster containing 2.4 GHz dual-CPU / dual-core AMD Opteron DP 280 compute nodes with 4 GB of memory, running a Linux 2.6.18 kernel on an XFS filesystem using an 1 Gbps Ethernet network, each with a WD Caviar RE, 7200 RPM harddisk with 16 MB cache.

A test setup is created, consisting of 3 MMW processes, each running on a separate node in the cluster. Agents are injected into the Mansion system, transferred to the first MMW, and subsequently transferred through two additional MMW processes before being retrieved by its owner. Timing results are taken at each of these nodes. The ALS is configured to use AOS for communication, and run on a different machine than the MMW processes.

All tests use a modified AOS written in C++ that include microsecond timers. The MMW is written in C, and a SunRPC dispatcher is used to invoke methods on AOS in these tests. Internal to AOS, all connections are configured to use 128 bits AES encryption with SHA-1 message authentication.

Tests of the ATP use ACs of 3 sizes: 500 KB, 1 MB, and 5 MB, respectively. Segments in the AC contain 5120 bytes of random data, with the 500 KB AC containing 100 segments, the 1 MB AC containing 200 segments, and the 5 MB AC containing 1000 segments. The tests are run up to 7 times for each AC size. The measurements selected for this paper are median measurements or close to average. Outliers are observed in the ATP tests. Inspection of the MMW log files shows that in these cases, concurrent activity took place in the MMW, for example, a ToC signature is received and verified in the MMW, while AOS was busy unzipping an AC, corresponding to the concurrent steps 5 and 6 in the Mansion ATP protocol outlined in Sec. 8.3.2. As the MMW and AOS are concurrently running processes which are multithreaded by design, some interference is inevitable for measurements of the Mansion ATP protocol in a live system. However, as such outliers do not represent pure AOS performance, median values are chosen to suppress the effect of those outliers.

	500KB	1MB	5MB
create ToC	8.4	9.1	14.9
sign ToC	7.7	8.1	11.7
zip AC	34.2	66.4	321.9
sync AC	21.7	36.3	102.7
total	87.6	127.5	450.2

Fig. 36. Breakdown of finalize cost (in milliseconds) for ACs using the C++ kernel. Results for the run with median total cost.

8.4.1. Finalize costs

This section repeats the measurements of Sec. 4.7.2, on the machine used in Sec. 8.4.2, to allow for comparison of the results.

Fig. 36 shows a micro-benchmark of the finalize costs of agent containers of 500 KB, 1 MB and 5 MB containing random data. As with the results in chapter 4, ToC checksumming and signing cause little overhead, even for large ACs. Creating a zip file and sync'ing it to disk again cause substantial overhead. Zipping overhead is nearly linear to the total AC size. *Sync* is again rather expensive.

8.4.2. Overhead of the agent handoff (ATP) protocol

This section describes the performance of the Mansion ATP protocol outlined in Sec. 8.3.2. These tests measure the ATP overhead after an agent has migrated one hop; the receiving MMW must verify a two-level audit trail.

	Protocol step	Time (msec)		
		500KB	1MB	5MB
S	Finalize AC	87.6	127.5	450.2
S	MMW sign ToC	7.7	8.1	9.0
R	AOS extract AC + verify ToC	25.7	215.0	565.6
R	MMW check toc signature	0.8	1.4	2.5
R	Audit trail verification	1.6	2.6	4.2
S	AOS ship_ac completion	68.4	265.2	636.4
S	MMW-level ATP completion	171.9	350.6	726.7

Fig. 37. Performance of an AOS-based agent transfer protocol (ATP) with agent containers of different sizes. S and R indicate AC sending, resp. receiving side.

Table 37 shows timings for the most important steps in the Mansion ATP. It depicts the total time it takes to ship an AC of 500 KB, 1 MB or 5 MB consisting of segments of 5 KB containing random data, as well as some of this operation's component costs.

The overall time for the ATP to complete for a 500 KB agent container is 171.9 msec. For an 1 MB AC, the shipping time is 350.6 msec, and for a 5 MB AC this is 726.7 msec. This time does not include the finalize time (taken from table 36), but does includes channel setup, shipment of the zip file, receipt, extraction, and verification of the AC and the audit trail at the receiving side, verification of the returned ToC signature, and committing the ALS update.

Signing and verifying ToC signatures requires public key cryptography. As can be seen from table 37, the overhead of these operations, as well as for audit trail verification, is

negligible compared to the overall migration cost. The overall migration cost is dominated by zip and unzip times. Unzip times are the major component of the AOS extract AC and verify ToC measurement shown in table 37.

AOS-level AC extraction and ToC verification times are not completely linear with respect to the AC size. The complete AOS *ship_ac* call only returns if the receiving side has received, extracted, and verified the AC. Therefore, the AOS *ship_ac* completion measurements are dominated by AC extraction cost. The overall MMW-level ATP completion time is somewhat longer than the AOS *ship_ac* completion time which it includes. This is caused by the additional interactions required at the middleware level, compared to the AOS-internal interactions.

The AOS ToC design is optimised to make efficient (binary) comparison between ToC entries of different ToCs in an audit trail possible. As can be observed from Table 37, audit trail verification is indeed efficient: it poses a negligible overhead compared to the overall overhead. Between 1.6 and 4.2 msec is measured for a 2-level audit trail, depending on AC size. Note that for ToC comparison, only access to ToC segments is required, not to other segments. This is because correspondence of the segments in the AC to the ToC entries of these segments has already been verified by the AOS kernel as part of AC integrity verification.

Table 37 shows that the Mansion ATP can be implemented with little overhead compared to the basic cost for finalizing an AC and transferring it to another AOS kernel over a secure AOS channel. The major cost component is zipping and unzipping the AC, and *sync*'ing the AC to disk, as was already reported in chapter 4; as indicated there, these cost components can be optimised away.

In conclusion, the ToC and audit trail verification mechanism is very efficient and causes negligible overhead comparable to the cost of shipping ACs, even with a 5 MB AC that consists of a very large number of segments. Further (end to end) measurements of agents migrating in a complete Mansion system, including agent transfers and agent management, will be presented in chapter 10.

8.5. Putting things together: the MMW implementation

This section summarizes how the MMW process is internally organised. The core of the MMW is an internal **agent table** with one entry per agent, and a number of RPC services that implement and provide access to middleware functionality.

The MMW consists of the following RPC services:

- The *MMW_ATP* service that waits for incoming agents;
- *SimpleComm*, the service invoked by remote MMW processes to queue incoming messages on interagent connections;

- The *MansionAPI* service that implements the MansionAPI calls described above;
- An *RPC forwarder* service. An RPC service is created for each object binding, which forwards the marshalled RPC/object invocation calls to the appropriate object server for invocation. There is at least one RPC forwarding service endpoint per agent, possibly also different RPC forwarding services for different object bindings multiplexed on it. The RPC forwarder service is itself an RPC service.

The ATP service is started by the MMW's main thread, which also registers the ATP services' endpoint in the Mansion location service. The ATP service uses an initialised AOS kernel[56]. As soon as an ATP connection is made, the service can create an ATP endpoint on AOS to wait for the AC. After receipt of the AC, a (temporary) entry is created in the **agent table**. The agent table is similar to a process table in an operating system; it contains all relevant information about the agent so that it can be looked up easily by the MMW. The agent table contains metadata such as the agent's AgentID and AgentOwnerID, a list of open connections, the (jailed) agent's local process ID, its jailing directory, and once the agent runs its PID (process ID) and the PID of its jailer process, its status (whether it is running or not), etc. This information is filled in when the agent's AC is received, verified, and started up. The target room in which the agent is to be started up is specified by the sending MMW, which sent this information as part of the Mansion agent transfer protocol; this information will also find its way into the agent table.

Initially, when the agent table entry is created as an agent arrives, its table entry is locked. The agent's status is set to *MIGRATING*, which ensures that incoming (re)connection request for the agent are returned the corresponding status; the status may be kept *MIGRATING* in the local MMW, even if the ALS' status is already updated to *RUNNING* due to completion of the handoff protocol. The MMW sets the status to *RUNNING* only after the agent process has started and all bookkeeping is completed.

Besides verifying the audit trail as part of the initial handoff protocol, the MMW has some work to do. It re-instantiates communication channels using information contained in the AC (i.e., creating an entry for each open connection in the agent table, and creating a ring buffer for incoming data for each of them and and filling it with any received, unread data found in the agent's AC if applicable). Also, a private jailing directory is created for the agent. Also, before accepting the agent, the MMW should check for reachability of the RMO that is specified in the migration information.

If all preparations are completed, the MMW can start the agent in a jail. It prepares a jailer startup argument string to pass to the *execve* of the jailer program, after forking. This startup argument contains the jailing directory name, and the contact address of the RPC endpoint of the MMW's MansionAPI service.

[56] The MMW can find the AOS kernel using information provided when started up; typically, AOS is started up just prior to the MMW, except if a single AOS kernel is shared between multiple middleware processes.

Before startup, the MMW creates a credential in the agent's private jailing directory (this directory cannot be read by other agents, since these are also jailed) using which it authenticates to the MMW after startup. The MMW then checks the agent table to verify that the credential matches the one from the started-up agent to avoid that arbitrary processes connect to it in place of the intended agent[57]. After the agent is started, the MMW notes the jailer *PID* in the agent table; the MMW can send signals to the jailer to *kill*, *suspend*, or *resume* the jailer including all agent processes in it.

After startup, the first thing that an agent does is call *bind_rmo*. Although the RMO to bind to is already known—the handle of the RMO is specified before migration and cannot be influenced by the agent—the agent needs to call *bind_rmo* to create a forwarding endpoint for the RMO binding in the MMW; *bind_rmo* returns the MCR of this endpoint, such that the agent can instantiate its RMO interface. The agent can then use the room it has entered using the MansionAPI and RMO interfaces. Only after *bind_rmo* is called successfully, is the agent's state in the agent table set to *RUNNING*.

8.5.1. Error handling and the Morgue

If an error occurs before setting the state to *RUNNING* and after the handoff protocol, or if an error occurs in general, the agent is shipped to the Morgue. The protocol for sending the agent to the Morgue is similar to regular agent migration, except that the Morgue unregisters the agent from the ALS and stores the AC for later retrieval by the agent's owner. Similar to cloning, the sending MMW invokes a method called *morgueify* on the Morgue service (like the WED, implemented as an object) to make it wait for the AC. After successful return of the morgue's *morgueify* method, the MMW removes the AC from its AOS kernel. It is the morgue's responsibility to register the agent's status as *DEAD* in the ALS, or to remove its AgentID from the ALS.

In the current implementation, after receiving an AC, the Morgue extracts it and verifies the audit trail on behalf of the agent's owner; it places the audit trail verification log into a directory that can be accessed by the agent's owner[58]. The resulting directory is created in a *MultiFileContainer* object that is accessible by an authenticated client program of the agent's owner (the *AgentOwnerID* is the ScID that is used to authenticate the client program). The Morgue provides a (private) directory per user, where all morgue-ified agents of this user are

[57] Currently, the MMW generates a new key pair using which the agent can connect to the MMW, using the ScID-based authentication protocol; this results in an encrypted SSL connection, which is unnecessarily inefficient. The *openssl* library makes it possible to revert back to the underlying TCP connection after authentication, which we do. An alternative could be to simply generate a random string and let the agent present it.

[58] AC extraction is a service to agent owners, which avoids that they need to be aware of the internal AC layout and which makes it possible for them to directly retrieve the AC's segment (currently, using a *MFC* client program). Audit trail verification is efficient and can takes place as an extra service at little extra cost. The (signed) ToC audit trail components needed for audit trail verification are kept with the AC so the agent's owner can redo the verification if needed. The Morgue provides a human-readable output log of the audit trail verification that the agent owner can check. If an error is found in the audit log, the audit trail's ToC and signature components can—if needed—be presented to the world owner or used in court as a means to take action against the MMW/zone member/zone where, according to the audit trail, something went wrong.

stored. All files/segments, including TOC entries and signatures and keys of all hops of the agent's itinerary, are available in the morgue's agent download directory. Thus, the user can verify the audit trail independently if needed.

8.5.2. The MansionAPI, binding, and RPC forwarding

The MansionAPI is a service that implements several mechanisms. These can be summarised briefly based on the information provided above and in previous chapters.

One part of the API deals with interagent communication. As soon as an agent sends or reads from a connection, the corresponding peer's status, peer agent's SimpleComm endpoint, and/or read buffer status is checked in the connection list in the agent table. If applicable, the peer endpoint is resolved using the ALS. The corresponding method is completed either by calling the remote SimpleCall method, or by reading in data from the local read buffer or blocking if not available.

Agent migration has been described extensively earlier in this thesis. The essence is that a check is made if a suitable MMW corresponding to the hyperlink's (target RMO's) object handle in the current RMO can be located. Next, the invoking agent is suspended (by sending a signal to its jailer process), unwritten or modified files from the agent's jailing directory and unread communication buffers are placed in the AC, and the AC is finalized using an AOS call. Next, the agent transfer protocol described in Sec. 8.3.2 is invoked. This either results in a successful handoff, after which the target MMW becomes responsible for managing the agent, or in an error, in which case the agent is restarted (its AC, buffers, and jailed process were only locked, not deleted or killed yet, so restarting is a simple matter of resuming the agent and unlocking its agent table entry after setting the agent's status to *RUNNING* again).

If migration succeeds, the agent's process is killed, the agent table and communication buffers are cleared, and its AC is deleted from the AOS kernel. Any bindings, c.q. RPC forwarding services are cleared automatically (where relevant, error codes are propagated and underlying connections are closed) when the agent process exits, as the underlying TCP connection between agent process and MMW is closed. This implementation was chosen because this way a single approach suffices to also handle involuntary exits of agent processes due to errors, and resulting automatic closure of connections and bindings.

The final part of the API deals with bindings to the RMO and objects in the room. The MMW uses the RMO to obtain information about entities registered in the RMO, i.e., the agents, hyperlinks and objects which are in the room. This ensures that operations on entities in the room (such as binding to objects, or room-local connects) take place on entities which are indeed in the room. If a given entityID is not found in the RMO, or if it has the wrong type, the respective operation will return an error to the agent. As explained earlier, binding to an object is essentially the establishment of a forwarding RPC service which is internally coupled to the endpoint of the object in an object server where the object resides.

Except for overwriting the ScID field of an object invocation with the *AgentOwnerID* of the agent, forwarding simply involves forwarding the marshalled invocation to the RPC endpoint for object invocation of the appropriate object server. Obviously, return values should be forwarded back to the appropriate agent which invoked the original method on the object. Although the current implementation of the forwarding mechanism is somewhat complicated, the logic required in the MMW is uniform and rather straightforward. The forwarding mechanism is independent of the specific object or method type invoked as marshalled requests need not be modified except for the the first 20 bytes containing the ScID.

RMO bindings are established in exactly the same way as bindings to regular objects, except that these can only be bound to when an agent enters a room. The current room's RMO is registered in the agent table, and the MMW can check that bindings are made to non-RMO objects only in a straightforwardly way, as RMO's are registered as hyperlinks in the RMO only – implying that migration is allowed, but not binding – and they have a specific object type *RMO* indicated in the object handle.

8.6. Summary

This chapter provides an overview of the most important aspects of the design of the Mansion middleware (MMW), from a functional point of view.

The MMW manages all agents. It uses a jailing system to protect system resources (both local and remote) against rogue agents. The jailer makes the system independent of language-specific security mechanisms such as the Java virtual machine, although the jailer itself is operating system and architecture specific (system call numbers and arguments differ for different architectures and sometimes even per kernel version, for example in Linux, see chapter 6). An agent is a process confined in its own jail, along with possible child processes. Each jail has a private jailing directory, and it can access an endpoint of the MMW so that the agent can connect the Mansion API and the runtime stub libraries of (bound) objects. The MMW can directly control an agent in a jail. The MMW can suspend (and resume) agents, or kill agents. The jailer protects resources such as local user files, including those of the user that started the MMW. An effective protection system like the jailer is, in our view, a requirement to ensure in a simple, uniform (thus, language-independent) way that agents can be safely executed, which is one of the long-standing problems in the adoption of non-Java agents [50].

Agents come with files stored in their AC, which can be accessed through the Mansion API. Agents are compiled or dynamically linked with a library which contains stubs for the Mansion API and for the objects used in a world. An IDL and a simple data representation is used to compile (stub) interfaces in different languages. Stubs do not implement middleware functionality. Instead they make remote invocations on the MMW process. The MMW ensures that the agent can communicate with other agents and with objects to get its work done, without requiring complex and possibly vulnerable implementation of middleware functionality inside the agent's address space.

The MMW implements the Mansion API calls and establishes the internal communication routing required for bindings to objects, and for interagent connections. Access control lists consisting of ScIDs, corresponding to AgentOwnerIDs or some other (role or payment based) scheme, are used to govern access to rooms and objects. Agent migration is implemented by the MMW using a lookup of available MMW processes in a target room's zone, after which the agent is migrated. If migration is successful, the agent on the original host is killed; if not, it is resumed in its current room.

.

Chapter 9
The Mansion API

This chapter explains the Mansion **application programming interface (API)**. The Mansion API contains the methods that *agents* need to interact with and use. This chapter describes the details needed to understand the API and the mechanisms involved. The API is implemented by the MMW process using the middleware components discussed in previous chapters. The Mansion middleware process starts and manages mobile agents, and provides them with a runtime interface which contains the Mansion API (Sec. 8.2.8).

9.1. Methods of the Mansion API

This section describes the most important calls of the Mansion API. The Mansion API consists of calls needed to migrate by following hyperlinks, to communicate with other agents, to obtain status information from the MMW, to store information in its private agent container and to retrieve information from it, and to bind to objects.

Fig. 38 shows a simplified list of API calls, grouped by function. The following sections describe the calls of the Mansion API per category. A following section describes the RMO interface. A discussion on the implementation of the API methods in the MMW closes the treatment of the middleware design and implementation. The final chapter of this dissertation discusses applications of the framework.

Method	Description
Context:	
get_current_room ()	get current room ID (object handle)
get_room_parm (parm)	get value of some parameter set in RMO
Information about self:	
get_gaid ()	returns the agent's own global AgentID
get_hopcount ()	agent's hopcount (# of physical migrations)
get_parent (depth)	get gaid of parent 'depth' up the parental tree
get_clonedepth ()	how deep are we in the parental tree?
get_rusage ()	get information on resource limits and use
Migration:	
get_target_room (entityID)	get handle/RoomID of (target) room
follow_hyperlink (entityID)	follow hyperlink; migrate
jump (RoomID)	jump to room directly if allowed (see text)
Communication:	
connect_local (entityID)	connect by entityid
connect (gaid)	connect by global AgentID
accept ()	wait for connection; accept
send / recv (desc, bytes)	read or write bytes onto connection (stream)
close (desc)	close connection (descriptor)
select (desc_set[])	select (poll status) descriptor set
get_peer_info (desc)	get peer's gaid (if allowed) and MMW scids
Agent Container:	
create_seg ()	create a new AC agt_data segment
write/read_seg (seg, offset, bytes)	write / read data to segment
delete_seg (seg)	if not persistent, remove
set_persistent (seg)	persistent seg cannot be removed
keep_on_cloning (seg)	keep this segment in child
Objects:	
object_bind (objectID)	get RPC endpoint to initialise stub (see text)
unbind ()	delete endpoint
Cloning:	
clone ()	clone (if allowed) in current room

Fig. 38. Overview of the most important calls of the Mansion API seen by agents

9.1.1. Context

Context-related calls provide agents with a way to obtain their current context, in particular about their current room. Using this information, an agent can detect whether it is in a room that it entered before. Suppose a world is a graph. Then an agent could enter a room from different directions without realising it was there before. Normally, an agent will determine the target room's RoomID before migration, either from information in the AS or by using the *get_target_room* call (see below), however the possibility cannot be excluded that it obtained incorrect information.

An agent may store the object handle of each room it visits in a segment in its AC. This is important when an agent has to search through a lot of rooms in a large and possibly complex world.

Get_current_room returns the object handle of the room's current room. The object handle is a world-wide unique and acts as a unique identifier for the room.

Get_room_parm returns fixed parameters defined for a room; a parameter is an attribute=value string. An example is *ISCONFINED=1* for a confined room.

9.1.2. Information about self

Information about *self* is useful when an agent wants to interact with other agents. An agent should know its own global AgentID, so that it can give this AgentID to other agents. For example, an agent can place its AgentID in its attribute set in its current room, or possibly in a white or yellow pages service. An agent can find its AgentID using the *get_gaid* call. If an agent does not explicitly announce its global AgentID in an AS, other agents can still connect to it using the *connect_local* call that takes the room-relative *entity_id* as an argument (see below); if necessary, agents can then exchange their AgentID for later use.

Get_parent and *get_clonedepth* are important for applications where an agent clones itself. For example, multiple child agents may search certain parts of a world in parallel to find a given piece of information. In this case, the child agent can find out *if* it is a child (if its *clone depth* equals 0 it is not), and if so who its parent is, up to its top-level ancestor. A conceivable scenario is where the initial agent takes the role of a master which coordinates work, and where child agents can connect to it to obtain tasks. Another scenario is where agents communicate only with their parents one level up.

The *get_clonedepth* call returns the clone depth (depth = 0 indicates the top-level parent). The return value of the *get_parent* call is the global AgentID of the parent at depth N. *Get_hopcount* returns the number of times an agent migrated (using *follow_hyperlink* or *jump*) since it was injected or cloned.

The *get_rusage* call returns a document containing a list of resource limits (e.g., *time to live*) as well as resource usage statistics, formatted as a list of attribute-value strings. Although a number of resource parameters are fixed, additional global parameters by a world

may be defined by the world designer. These should be documented in programmer instructions.

9.1.3. Migration

Using *get_target_room*, an agent can acquire the object handle of the target room. Based on this, the agent can decide whether to move to the target room, e.g., by checking a list of already-visited rooms (see ''room context'' above). The target room's handle is also useful to derive which zone the target room is in (Sec. 3.3.4).

In some cases, an agent may want to avoid visiting certain zones—an agent owner may place information about untrusted zones in the agent's AC, which the agent can check the against the ZoneID in the object handle of a target room. Another way to use the ZoneID information in a target room handle is to construct a search pattern in which all rooms in the current zone are visited first, before following a hyperlink to a room in another zone. In other words, by being able to obtain the RoomID of a target room (which is available from the RMO), agents can optimise search.

If a world supports the *jump* call (see below and Sec. 3.8.1), an agent may also keep a list of interesting ''still-to-visit'' rooms, in addition to a ''visited-rooms'' list. This may facilitate searching a world efficiently.

Migration calls are *follow_hyperlink* and *jump*. Using the *follow_hyperlink* call, an agent follows a hyperlink to a room. If the target room is in another zone, the agent has to be moved to a MMW process in that other zone, typically on another machine. Agents have to ensure they store any relevant information in their Agent Container before they call *follow_hyperlink*, which finalizes the AC. If following a hyperlink is successful, the agent is killed and restarted in the context of the target room. Otherwise, it is restarted where it left off, with a return value indicating failure of the *follow_hyperlink* call.

Jump takes a RoomID as an argument. If allowed, then, an agent migrates to the target room, just as if it followed a hyperlink. *jump* was not considered in the original Mansion design, and it is not always allowed. It can be disabled completely using a setting in the world design document. There are also cases where hyperlink constraints (chapter 10) will cause *jump* to fail, returning an error and restarting the agent similar as to what happens when *follow_hyperlink* fails. This may happen if a hyperlink constraint defines that the agent's current zone may not link to the zone of the target room which was specified as an argument to *jump*. Hyperlink constraints apply to all types of agent migration. Note that the underlying mechanism for *jump* is identical to that of *follow_hyperlink*, only the argument differs (*RoomID* instead of the room-relative *EntityID*).

A few notes on jumping are useful. By disallowing jumps, it is possible to enforce structure on an agent's migration path, to force it to follow hyperlinks that form some predefined path. This seems a useful constraint: there may be logical, commercial, or security reasons for imposing structure on an agent's migration path. An argument in favour of jumps is that

an agent may need to search through a large number of rooms, and may need to back-track to ensure that it reached and searched all rooms.

Backtracking 10 rooms to get back to an earlier room (if it is even possible to find the way back given a particular world topology), implies following 10 hyperlinks and if not 10 physical migrations, at least 10 agent process restarts, which is very inefficient. *Jump* allows for skipping these steps and improving efficiency by simply providing the identifier of the room to jump to. The application of zone-based hyperlink constraints to jumping makes it possible to still enforce structure on worlds, although creating fine-grained structure *in* the world, i.e., by having zone administrators place hyperlinks in specific ways, is no longer clearly possible. Clearly, jumping has its pros and cons. For that reason, it is up to the world designer to decide on whether jumping is allowed.

It is conceivable that Mansion could be extended such that the room from which a jump is made can define a policy that disallows jumping (possibly by setting a parameter in the RMO), or a more specific policy which (dis)allows jumping to a specific zone. Also, the zone to which a jump is made can make *inbound* checks: it can verify that the agent comes from a zone from which a hyperlink may be made to the particular room that the agent wants to enter. In other words, it may define a **zone entrance policy** based on the zone where an agent came from.

Note that every zone already has one or more zone entry rooms (ZERs). These are rooms that are linkable to from other zones. Normally, the default ZER is the first room created in a zone (with RoomID ending in RMO_0_0); other ZERs can be added to as part of the zone description in the zone list in the basement. Only a ZER may be linked to from outside the zone. Other rooms are simply not reachable from another zone, either by linking or by jumping. This avoids the problem of ‘‘deep linking’’; deep linking is a known problem for some Web sites in the World Wide Web. For example, newspaper Web sites may want to avoid that anyone can link to their articles without first going through the front-page, which may act as a ‘‘pay wall,’’ or which may contain advertising to similar effect. Zone entrance policies are a generalised form of the above policy, which can be applied with and without jumping (in fact, the receiving MMW cannot see the difference).

From the above, it is clear that jumping can be subjected to relevant security and topological constraints enforced at migration time. In all cases where migration is not allowed by policy, *jump* (or possibly *follow_hyperlink*, when a hyperlink refers to an erroneous *RoomID*) will return a specific error code to the agent.

9.1.4. Communication

Communication calls are provided through a socket-like interface. All agents have a communication endpoint which is always enabled. Agents use *accept* to accept incoming calls on this endpoint. *block* specifies whether an agent wants to wait for an incoming connection, or return immediately with an error code if no connection request was pending at that time.

There are two *connect* calls. Both have an optional *timeout* argument, One connect call takes an agent's *EntityID* relative to the room as an argument. This call is intended to allow an agent to make contact with any other agent in the same room. The other call takes a global AgentID as an argument. Local connections are closed automatically when one of the agents migrates to another room. Non-local connections set up using *connect* remain active during migration. Unread messages are stored in an agent's AC when it migrates to another room; after migration, the receive buffers are reinstantiated by the MMW. When an agent sends data and the peer agent has moved, the MMW will transparently resolve the peer agent's new contact address and reconnect. An agent is responsible for keeping track of the *ConnectionID*s of its open connections by storing them their AC. Connection identifiers for non-local connections are persistent until the connections are closed using the *close* call.

Send and *recv* do what their counterparts in BSD sockets do: send and receive sequences of bytes from their peer, reliably and in order. Errors are given as negative return values. *select* returns the status of the set of connection identifiers passed as an argument, e.g., to check whether a descriptor contains data to read.

Get_peer_info does not have an equivalent in BSD sockets. This call returns a data structure which contains the peer's AgentID (if allowed by the other agent. This is governed by a flag on the *connect* and *accept* call), and the peer MMW's PeerID and ZoneID (Sec. 3.2.1). The *options* argument of the *connect* and *accept* calls contains a flag that governs whether one's AgentID is visible to peers that are connected to. With *connect_local*, AgentID is never provided to the peer. Agents have to explicitly pass their AgentIDs if they want to establish a migration-transparent persistent connection.

9.1.5. Agent container

This subsection of the API contains AC related calls. These calls concern data segments only. Data segments are segments which are internally to AOS typed as *AGT_DATA*. Other segment types, like those containing MMW data, are not visible to agents.

Create_seg is used to create a new data segment. The agent can specify a *type* and a *name* string; internally to AOS, *type* is placed in an AOS *subtype* field. *name* is an uninterpreted character string. *create_seg* returns a segment identifier for subsequent calls.

Write, *read*, and *delete_seg* do what their name suggests: write and read data to and from a given *offset* of the segment, or delete a segment. Segments have semantics comparable to files. There is however no internal (implicit) offset and no corresponding *seek* call. Offsets must be specified explicitly as arguments to the *write* and *read* calls. A programmer can construct an agent library which maintains file descriptors with offsets and construct a library seek method, if needed. Creation of holes (by writing beyond *end of segment*) is not allowed and will return an error.

Set_persistent is an important, Mansion-specific call. Once called, the segment is marked as persistent in the AC, and cannot be removed without triggering an error when

verifying the agent's audit trail (see Sec. 8.3.1). Audit trail verification takes place in the MMW which receives an agent. If an error occurs, migration is aborted. This confines an agent to its current MMW, which either modified or removed a persistent segment, made an error constructing the audit trail, or hadn't verified the AC when it came in.

Keep_on_cloning is important for the *clone* call; when cloning an agent, only its code segments are retained, and any segments marked with *keep_on_cloning*. The procedure of cloning is explained below and example use is described in chapter 10.

The current Middleware implementation also copies an agent's data segments into a special directory in the jailing directory before the agent is started up, and writes modified files back to the AC. Accessing segments as files is very convenient, as system calls exist to conveniently and efficiently access files directly, and most programming languages come with libraries that allow file manipulation. Also, jailed (binary) programs can spawn other programs for file manipulation or filtering tasks; some binary agents (e.g., image analysis programs as outlined in chapter 10) consist of multiple programs, each performing a ''stage'' on an input file. Some scripts, input (configuration) files, as well as output files can be accessed directly on disk in the agent's private jailing directory. This way, many (legacy) programs can function without modification to execute an agent's task or part thereof.

New files written to the data directory are automatically added to the AC when an agent migrates. Modified files that correspond to a persistent segment are ignored: agents need to keep track of (or mark) persistent segments explicitly, as well as *keep_on_cloning* segments. If any of the direct AC manipulation calls are used directly, attempts to modify persistent segments will result in an error.

Note that the jailing directory is private. Agents can access only their own jailing directories. The above directory structure is only created for agent data segments which are not explicitly subtyped. Thus, agents have a choice to work directly with the API calls to access agent segments (if needed using a *type*), and in addition to use the directory convention outlined above for nontyped segments.

9.1.6. Objects

The *object_bind* call takes the *EntityID* of an object as an argument; if *EntityID* does not correspond to an object registered in the room's RMO, an error is returned. Otherwise, the MMW establishes a binding that results in an RPC endpoint being created by the MMW. This endpoint is returned by the *object_bind* call as a Mansion contact record (Sec. 5.1.3), to which a run-time stub for the object can be connected.

If an agent's runtime system and programming language allow, binding may result in automatic instantiation of a new runtime object or interface that is internally connected to the RPC forwarding endpoint created by the MMW; in a C language binding, the stub is a statically linked library which can be initialised using an MCR, and which returns a *BindID* such that multiple object bindings can be invoked using the same (C) library interface.

Unbind removes the binding and the RPC forwarding endpoint from the MMW. Invocations on this endpoint will now return an error. (Note that multiple RPC endpoints may be multiplexed onto a single TCP port/connection; each specific binding is indicated using the *index* field which is specified in an RPC *request* header, Sec. 5.1.3).

9.1.7. Cloning

The *clone* call, internally, follows the same procedure as regular agent injection. This prevents code duplication and ensures that the world entrance daemon is involved in the creation of clones, which is important for resource protection and security (Sec. 8.2.1). First, the cloning agent's AC is finalized. Next, the MMW contacts the WED, which provides a *clone_get_atp_endp* method. This method prepares an ATP endpoint on its AOS kernel, and returns this endpoint and a transaction ID (*xid*) to the invoking MMW. Next, the MMW invokes the method *clone_get_result* on the WED interface using the *xid* as an argument, which blocks until the clone operation is complete.

Internal to the WED, the *clone_get_result* method invokes the *wait_ac* call on its AOS kernel. Concurrently, the sending MMW uses the AOS *ship_ac* call to ship the AC to the WED. Please note that *ship_ac* does not remove the original AC; it only ships a copy of the AC. This property is essential to cloning as described.

Handoff and removal of the original AC (not needed in case of cloning) is always coordinated explicitly at the middleware (MMW) level. The middleware level protocol used is specific for cloning, but it makes use of the same AOS ATP calls and audit trails verification mechanism as the regular agent transfer protocol.

After receiving the agent, the WED first checks the AC's integrity using the audit trail verification mechanism explained in Sec. 8.3.1. The WED also verifies that it has constructed the original agent's agent passport: agents can only be cloned by the WED that injected the original agent. Note the importance: this is the only way a WED can do resource accounting or limitation, and verify that it knows the agent's owner, substantiate reversible pseudonym schemes, etc. The WED can keep track of counters—for example, a maximum clone count—for its agents.

After all the above integrity verification steps check out, the WED creates a new empty AC using its AOS kernel. Then the WED copies over all segments that the parent agent's AC *originally* contained, that is, when it was first injected; the WED can determine this by inspecting the first *ToC* of the original agent. Any segments marked as ''keep on cloning'' are also copied, and if relevant made persistent. Next, the WED creates a new *AgentID* for the agent by calling the ALS (Sec. 3.3), creates and signs a new agent passport for the child agent. It also adds a persistent segment which contains the child's *clone_depth* and the child's parent *AgentID* to the new AC. This information is relevant for the agent's ''information about self'' calls (see above). All this information is signed by the WED, by signing the new agent's first ToC. Next, the new agent's AC is shipped to the MMW that created the

clone, or to another MMW in the cloning MMW's zone; after successful migration, the ALS is updated to contain the agent's new contact address after migration. Note that this protocol is identical to the one used for injection, which in turn is the same as the regular agent transfer protocol outlined in Sec. 8.3.2. Following successful migration, the parent agent's AC is deleted from the WED's AOS kernel.

During the *clone* call, the parent agent is suspended by the MMW. The *clone_get_result* call returns only after the child agent is successfully migrated (or if an error occurs). Subsequently, the invoking agent is unblocked and receives an error code indicating success or failure of the clone call. If successful, the clone call returns the *AgentID* of the newly created agent to the parent.

Please note that the way in which the *clone_get_result* call is invoked by the MMW—as a synchronous method invocation that returns when the clone operation is fully completed—is an artefact of the fact that the WED, like all Mansion services, is currently implemented as a passive Mansion object; the object may not do anything, and has no thread active, in between invocations. Thus, the invocations on the underlying AOS kernel can only take place while the *clone_get_result* call is invoked.

The world entrance daemons are effectively the *root of trust* with regard to all agents injected into a world. The parent agent's WED is involved with cloning, which ensures that *a)* the new agent's AC and audit trail is reset to a pristine state, matching the parent agent's original state, removing all unnecessary data segments in the process, and *b)* the WED is able to enforce resource accounting and limiting with regard to the number of agents cloned by a given top-level ancestor, or of a given agent owner. Also notice that because Mansion disallows cloning by any party except known and trusted WEDs, the possibility that a world is flooded by rogue, unknown agents from unknown origin or with an unknown *AgentOwnerID*, is limited compared to the situation where agents could be cloned or injected by arbitrary MMW processes.

The *clone* mechanism is conceptually clear and straightforward to implement, but inefficient. An optimised version is described as future work in appendix 6.

9.1.8. The room monitor object interface

This section describes the methods of the room monitor interface (RMO interface). The RMO is accessible to agents in its room; it contains metadata regarding agents, objects, and hyperlinks in the form of attribute sets (ASes), which are accessible to agents. It also contains information which can only be accessed by the MMW, such as information that the MMW needs to connect to agents (*AgentIDs*), or the *RoomID* which is internally associated with a hyperlink. The information in the RMO is essential to the functioning of the MMW.

Below is an overview of the most important methods of the RMO interface.

Method	Description
Attribute set related methods:	
get_entity_list (type)	List all entities of a type (may be ANY)
get_entity_type (eid)	Return the entity's type
as_get_matches (type, template)	Return list of matching eids (0 if none)
wait_for_event (type, template)	Block until one or more matching eids
get_as (eid)	Get the attribute set of an entity
chg_as (eid, AS)	Change the AS to specified set
Middleware-internal methods (hidden from agents):	
get_entity_record ()	Get relevant entity information, e.g., AgentID, or the object handle of an object or hyperlink in the room
register_entity ()	Register information, such as AgentIDs or object handles in the RMO, and the entity's initial AS
delete_entity ()	remove entity and AS

Fig. 39. Overview of methods of the RMO object

When an agent enters a room, it is automatically connected to the RMO; the agent can invoke the attribute set related methods shown above. (Technically, to bind to the RMO, an agent calls *bind_object(0)* on the MansionAPI or its equivalent *bind_rmo*; this returns the RPC endpoint of the forwarding service to initialise its stub with).

Agents can invoke only attribute set related methods. These include methods to list all entities of a given type (*get_entity_list*). *type* can be *Agent*, *Object*, *Hyperlink* or *ANY*. The call returns a list of *EntityIDs*. The calling agent can specify the *entity_id* to start with, and a *count* of entities to return. EntityIDs are also returned by *as_get_matches* and *wait_for_event*, which take a *template AS* as input which is matched with all attribute sets in the RMO; matching entityIDs are returned. The difference between *as_get_matches* and *wait_for_event* is that the latter blocks until one of the attribute sets match. In all cases, the detailed attribute set has to be requested from the RMO explicitly using the call *get_as*. This call returns the requested attribute set into a caller-provided buffer.

Currently, the AS is formatted as a set of *attribute=value* strings separated by space characters, terminated by a NULL character, contained in an array of maximum 1024 bytes. The maximum size can be adapted by changing the RMO type definition. *Chg_as* is used to change an attribute set. The old attribute set is completely overwritten by the new one specified with *chg_as*. The RMO ensures that EntityID and EntityType attributes, set at entity registration time (see below), cannot be overwritten.

The *register_entity* call is made to register an entity, and its (initial) attribute set. The method takes an *entity type* as an argument, and a pointer to an AS (or NULL); the AS may not contain an EntityType attribute or an EntityID attribute; these are defined and set by the

RMO. An agent's initial AS is empty except for the *EntityType* and *EntityID* attributes. Only the agent (or its MMW) can modify or remove its AS.

The internal middleware methods revolve around entity records. The entity records in an RMO allow any MMW in the zone (also zone member processes where the agent does *not* run) to obtain information about the entities in the room.

For objects, the *entity record* contains the object's *ObjectID;* for agents, the global *AgentID* (recall that agents may or may not announce their *AgentID* in an attribute set; if not, the agent is only reachable using the agent's room-relative *EntityID* by means of the *connect_local* call. Internally, the MMW can look up the *AgentID* in the RMO to service the call). Finally, for hyperlinks, the entity record contains the *RoomID*, which is the object handle of the target room's RMO.

The hidden calls of the RMO interface are protected using the RMO's role bitmap. This ensures that only MMW processes in the RMO's own zone can call the methods for registering and unregistering entities in the room and for obtaining the entity records. The internal information about an agent is thus not visible to other agents, but available only to MMW systems in the same zone.

9.2. Discussion

This chapter provides an overview of the Mansion API and the room monitor object interface, and describes the functionality of the different methods of these interfaces. The MMW implementation brings the different components and pieces described in earlier chapters together. This chapter describes the way in which the Mansion middleware (MMW), to implement the Mansion API calls, uses other components of the world, particularly the world entrance daemon and the ALS (particularly, for agent injection, cloning, and the morgue).

The MMW is a program that contains an agent table and provides services: the MansionAPI and RPC forwarder service to agents, and the agent transfer protocol (ATP) and SimpleComm services to other MMW processes in the outside world. Mansion uses the jailer, by means of which agents are confined so that they can, by default, only access a private jailing directory. This directory is also used as a means for agents to access data segments from their agent container, allowing agents to reuse (legacy) file manipulation libraries or programs without having to use the segment creation and manipulation routines provided by the Mansion API. The RMO has been discussed, which provides methods to search the content of a room effectively.

The importance of the RMO is that information about entities in a room can be shared between all members (MMW processes) of a zone, so irrespective of where an agent in the room is running. We also described cloning, the approach used to create new agents with (a subset of) their parent's segments, and morgueifying in detail.

Summarising, this chapter provides an overview of the most important calls of the MansionAPI, with details on implementation, where needed referring back to previous chapters

for details. In all, this chapter provides a broad overview of the overall functionality and implementation of the Mansion middleware system, the MMW in particular.

The description in this chapter is based on the Mansion middleware as currently implemented, which includes the MOS and all objects and services described in this thesis, as well as the jailer, AOS, and the MMW process as described in this chapter. Although implementation issues and some bugs remain, the system has been used to run worlds that demonstrate that the paradigm and the MansionAPI as described so far actually work. The following chapter describes some of these applications and our experiences in using prototype Mansion applications.

Chapter 10
Applications and Experiences

This chapter describes a prototype Mansion world with different agents, and our experiences. Strength and limitations are explored and discussed. The chapter also provides end-to-end performance measurements to indicate where most of the time goes when running mobile agents in the current Mansion prototype. The chapter ends with suggestions for improvement and a discussion.

10.1. A medical imaging world

This section discusses a world in which agents can search medical images. Agents can search raw images, using an image analysis algorithm implemented in a legacy (C) image processing program. The agent used in the prototype makes use of a standard program and library often used for medical image analysis [85]. The specific images stored in our world are functional Magnetic Resonance Imaging (MRI) images.

The world provides two types of rooms: confined and unconfined rooms. The core rationale for using confined rooms, is that medical imaging data is personal information from a legal perspective, and particularly privacy sensitive as it concerns health related information. This means that (without explicit consent from the patient) the images may not be shared with people outside the scope of direct medical treatment or, if applicable, outside the scope of a particular scientific experiment for which consent was given [73]. Without consent, the data may generally not be shared or stored in a data storage or processing infrastructure outside the hospital's domain or control [80].

The goal of using the fMRI use case is to show how Mansion can be used to allow data owners to share information such that it can be easily found and searched without the data leaving the hospital perimeter if the data is privacy sensitive. This demonstrates the use of confinement and addresses some of the research questions posed in Sec. 1.5. The prototype

world is one in which binary data is stored in different administrative domains, hospitals. Each domain is a zone containing rooms. Each room can be found using a straightforward hyperlink topology.

10.1.1. The MRI use-case: searching sensitive medical data

Magnetic resonance imaging (MRI) is an approach to medical imaging where, using a set of superconducting magnets (helium-cooled, making the equipment huge and extremely expensive), human tissue can be scanned 3-dimensionally at a very high resolution in a noninvasive way. No surgery is required, and in contrast to CT-scans, the scanner does not use ionising radiation. Using MRI scans, structural features or artifacts *inside* human tissue can be detected by scanning through a computer-generated stack of 2-D images. The typical use of MRI is to scan the brain images for visually or algorithmically discernible artifacts.

Functional MRI studies are specific neurological studies that make use of *time-series* MRI scans of the brain. MRI scans are taken of the brain over time while the subject executes certain tasks, or is at rest. This allows researchers to discover which parts of the brain are activated due to, for example, motor or cognitive stimulation [85].

Raw MRI data output by a scanner and stored in the hospital's data storage system does not contain an image of only the brain. It also contains high-resolution information about bone and skin tissue. From this information, a detailed 3D reconstruction of the subject's face can be reconstructed. Therefore, MRI images are considered personal, identifiable information.

An example MRI scan from an fMRI time-series is used for our experiments, with consent from the subject. The approach outlined in this section is applicable for any (MRI) imaging data, not just functional MRI.

Hospitals typically have huge data archives with medical images, stored in a picture archival system or PACS. The size of (f)MRI data is large—with older scanners, about 500 MB to 1 GB per fMRI time series, and with newer MRI scanners with a higher resolution, up to 1TB per scan. The fMRI scan used in our prototype application is of the older variety. The size of the picture archive makes it infeasible to export scans to a client machine on request, even if privacy issues would not exist.

There is a case to be made for sharing raw medical (MRI) data. For example, a researcher may be interested in patients that have a particular type of tumour in the left frontal lobe of the brain, for inclusion in a (retrospective) study or a trial. Finding a sufficiently large number of patients to be able to do research and obtain statistically relevant results is currently often very difficult—and expensive. Especially with rare diseases which occur only for a small number of patients spread over a large geographic area, possibly spanning multiple countries, it may be impossible to find sufficient patient (data) to do research on. Similar applies for enlisting patients for clinical trials. Approaches that help finding suitable patients more easily can help. Simply sharing medical imaging data over the network, however, is

infeasible both from a technical (security) and a legal (privacy) perspective.

Currently, finding patients for a trial—particularly internationally—is labour intensive and costly. The first problem is to find patients suitable for a particular study or trial, the inventory phase. This phase typically involves sending a request to all potentially interested hospitals, which contains a description of the research. In each hospital, the request is first passed to a medical-ethical committee, which has to agree with the goal and general approach of the study. If this check is passed, a detailed inquiry may be submitted. This involves asking the hospital on whether they have suitable patients and, if so, how many. The amount is important, particularly for pharmaceutical trials which are expensive and impose quite some (administrative) overhead [88]. For this reason, there often there exists a minimal number of patients that a particular hospital must provide to participate in a trial. The third phase may imply contract establishment or agreement at the researcher/hospital level, after which patients are asked to consent or participate.

Note that for a retrospective study, on the basis of patient records or data alone, the situation may differ slightly as here, any matching patient (record) may be eligible given that the record matches certain qualitative criteria and cost aspects may be less relevant. Still, overall the phases are similar.

Many aspects of the selection process involve manual search and checks to take place, often by doctors or researchers in a hospital, who may need to manually consult (written) patient records to assess suitability of possible subjects. As this is labour intensive, hospitals may consider it questionable whether the research is worth the effort. As a result, finding a group of patients with a specific disease in sufficient numbers to do valid research with (in particular, valid statistics) is often infeasible.

This chapter describes a scheme that concentrates on the first phase of trial selection, which in practice is often very cumbersome:

How to find patients with a rare disease, if these patients are distributed all over the world, while the patients' data may not leave the hospital premises?

The solution is based on confinement: agents of researchers can search data locally, *after* which the data holder can decide whether research collaboration is useful, based on the results of the search.

10.1.2. Example use case and approach

Consider a researcher who wants to explore the occurrence of a particular type of tumour. This tumour has specific characteristics that can be discerned in MRI images; he has developed an algorithm that can recognise a tumour in an MRI image with a high degree of certainty. The algorithm is implemented in an (open-source) fMRI analysis program called *fsl*[59].

(f)MRI images are nearly always stored in a hospital's PACS system. Note that for rare diseases, textual patient records often do not mention the tumour, or they may contain a misdiagnosis or imprecise descriptions. Image search may be the only way to find patients with a given disease, if this disease is rare enough. A similar scenario may hold for other data, such as CT scans or DNA sequences.

In the first phase of research, the researcher wants to *find* patients with the tumour in which he is interested—irrespective of where they are. Second, the researcher wants to set up a collaboration with the hospitals that treated these patients, to do a retrospective study, for example, to discover if patients have characteristics in common (e.g., environmental aspects, other diseases, genetic predisposition, etc.). Third, a trial or follow-up study may be set up. This may possibly involve other information about the patient, once the patient is enlisted. Only after agreement is reached on the terms for collaboration between the searching researcher and the data holder, will patients be asked to consent to sharing their information with the researcher, or possibly to participate in a trial.

Using confinement, the first problem can be solved: finding patients suitable for a trial or retrospective study. Mobile agents can search through raw, un-annotated medical imaging data to find matching patients. Every hospital provides a a confined room in which one or more objects reside that contain the raw imaging data (MRI scans) from the PACS system.

Mobile agents that are confined cannot export any data to the outside world, except by writing it to an *export file* (Sec. 8.2.15), which is sent to the confined room's guardian agent or its owner after the agent exited. The confined room's owner (or the *data* owner, physician) can evaluate the results and the *research request* and decide on whether to participate in the trial or retrospective research. To do this, the data owner can use the information (search results) placed in the export file by the agent of the requesting researcher. This file may for example contain the number of patients that matched the search, and perhaps additional information. For example, for each matching patient, a number may be given indicating the probability that the match is correct, or a thumbnail image may be generated by the agent, showing the tumour, allowing for visual inspection by the hospital as a check that the matches are correct.

Note the critical characteristic: the data owner obtains the results, not the requester. In contrast to current practice, using confinement, a clinician will receive a request for setting up research or a trial, *together with concrete, specific search results* including the (number of) patients that are found to match the request based on an algorithm implemented by the researcher. Hospitals, physicians, and medical ethical committees can now evaluate whether they wish to partake in a trial not just based on a general research description, but on concrete results of the customised search that was conducted *on-site* by a confined agent.

The mobile agent-based search is privacy-preserving: only when the requested hospital agrees to participate in a trial or (retrospective) research, will information be returned to the requesting researcher(s). Otherwise, not even the number of matching patients is returned. This is important, since confinement is vulnerable to leakage of information. Since the data

[59] http://fsl.fmrib.ox.ac.uk/fsl/fslwiki/

owner does not control how searching takes place, returning *any* information, even a single bit ("your agent found interesting information, please contact the confined room's owner") can be enough to leak sensitive information—for example, whether the prime minister who was admitted to the hospital recently had a brain tumour (note that the latter may even be found out using ''anonymised'' image data, if an observer can infer the time or ordering in which images are added to the archive).

The power of confinement lies in being able to search raw data, such as images and DNA data, directly—and possibly in combining this information with textual patient records containing diagnoses or medication information. The agent's search results are only visible to the hospital, and filtered by (manual) inspection of both the research request and the search results. This way, a human check is included on the (origins of) the request and the likelihood that the request is legitimate. This limits the chance that a malicious agent implements a highly tuned search algorithm to find out information about one or a few specific patients in the hospital.

Another way to address this issue would be to remove all identifying features from images provided by agents (a process called de-identification), but such efforts are often error-prone and not always feasible; for example, it may be harder to orient 'brain-extracted' MRI scans to a template than to orient (register) scans that contain facial features. The prototype example world does contain "anonymised" MRI scans but only for convenience of testing. For reasons outlined elsewhere [80, 73, 76], real systems should avoid data anonymisation as a means to ''secure'' data sharing as much as possible.

Besides a need for confinement for privacy reasons, there are other reasons for using mobile agents to search data locally. Efficiency is one: it may be easier to do high-performance computation at the hospital side, where a (small) cluster may be available, than at the client side, where a researcher may only have one PC available.

Adaptability is another reason for local search: agents may be able to search disparate records and combine results. Autonomous agents are often proposed as a suitable approach to implement adaptive search strategies, where components can be plugged into the agent for tasks like translation of terminology. The assumption is that it is easier to adapt or update agents flexibly, than it would be to index data or to modify a (remote) service's implementation [28]. In short, using customised agents to search data locally is itself of value: agents can implement ''adaptors'' for interfacing with different systems, and (local) translators can help find specific information in patient records encoded in nonstandard ways.

The solution presented in this chapter, is to search through raw, un-annotated medical imaging data, at the place where the data resides. This helps for efficiency, allows for a specific search highly customised for a given research question, when the raw data cannot be effectively be de-identified. Confined rooms can implement large-scale distributed search while protecting against leakage of sensitive content. The next section describes a world set up for this purpose.

10.1.3. A world for confined and unconfined MRI data search

This section describes a prototype world designed for searching medical data in different hospitals, shown in Fig. 40.

Each hospital provides one or more rooms, and the rooms of the hospitals are linked to each other in a predefined way. Every hospital has its own zone, which is registered in the world's zone list. Every zone has a zone entry room (ZER), that can be linked to. The ZER has one object containing de-identified MRI data, and a hyperlink to a second room that contains raw, identifiable MRI data. The latter room is confined. The unconfined images are ''brain extracted'' images that allow agents to ''scout'' whether a hospital may contain interesting patients matching a query, before entering the real search at the possible expense of hospital (manual labour) time.

Each ZER also has a hyperlink to the next zone called *nextzone* (by means of an attribute *name=nextzone*), forming a linked list. It is straightforward to add new hospitals (zones) to the list. Newly added zones (hospitals) are added at the beginning of the list: the world entrance room can link to the zone entrance room of the new zone, which can link to the room that was initially linked to from the world entrance room.

Navigation in this world is trivial: agents follow the *name=nextroom* links to visit all zones (hospitals) in the world. The simplicity of the scheme illustrates the strength of Mansion: for this application, having a linked list of rooms (hospitals) is the only thing needed. Why make it harder? If the world would become really large-scale, different (overlay) hyperlink structures may be conceived to help agents navigate. For example, different orthogonal hyperlink structures may group hospitals of different geographical regions (e.g., US, EU), or group hospital rooms with distinct content types (e.g., CT scans or DNS data). Possibly, each grouping may internally have an similar linear linked hyperlink structure similar to used in the prototype. Such sub hyperlink structures can be found using hyperlink attribute sets.

The confined room containing raw, identifiable MRI images and possibly other information is called *scans-raw*. The hyperlink to this confined room has a *name=scans-raw* attribute, and an attribute *confined=yes* (see Fig. 40). This indicates to agents that the hyperlink is to a confined room.

Fig. 40 shows the world described thus far. Every room has an AS describing itself, associated with the *self*-hyperlink *EntityID=0*. Before entering a confined room, an agent may make a transient note ''entering confined room'' in its AC. Using this attribute, an agent which finds a note ''entering confined room'' in its AC, can determine if it is currently in the confined room or back in the original room using the *get_room_parm* call. If back, it can remove the mark. Note that an agent in a ZER can also have a clone enter the confined room. The parent can then move on. The clone can self-destruct when done (see Sec. 10.1.6).

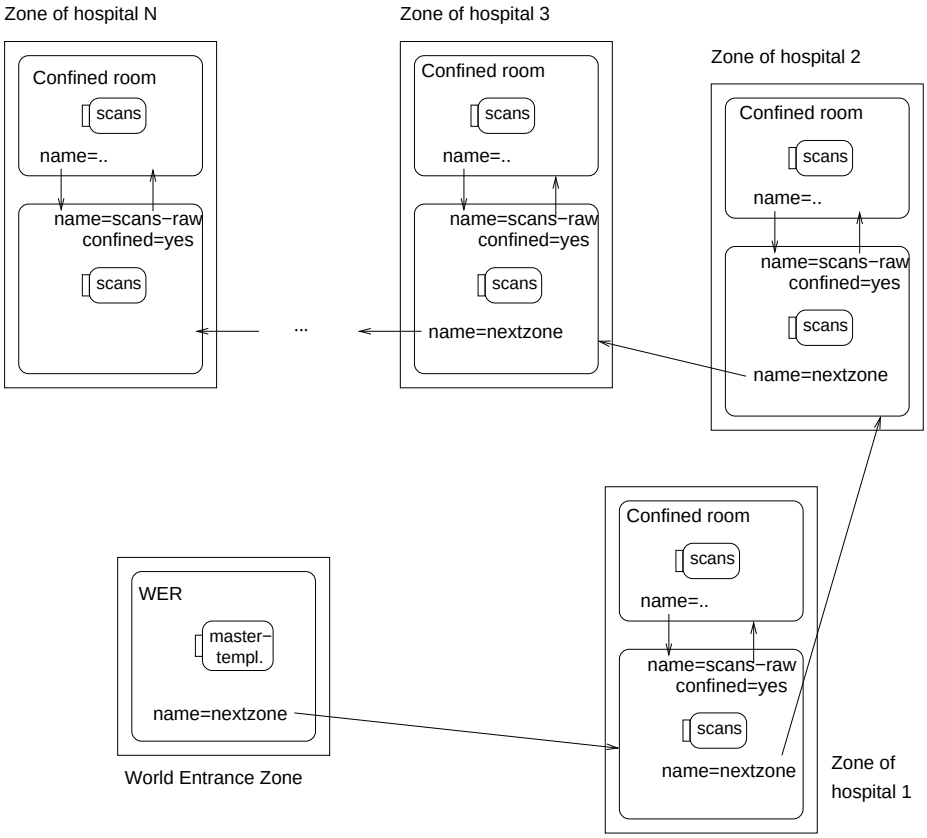

Fig. 40. Overview of the prototype MRI world. Hyperlinks with attribute ‘‘name=nextzone’’ links refer to the next zone; a ‘‘confined=yes’’ attribute indicates that a hyperlink points to a confined room (in the same zone); hyperlinks with ‘‘name=WER’’ can be created to point back to the world entrance room (WER) and vice versa (not shown). Confined rooms contain raw MRI scans, and possibly other objects interfacing with, for example, patient records. Regular rooms may only contain de-identified information such as brain-extracted MRI scans.

For navigating the world in Fig. 40, a few simple conventions suffice. These are directly encoded in the *name=* attribute: *name=scans-bet* (for objects containing de-identified scans) and *name=scans-raw* (for a confined room that contains an object with raw non-de-identified MRI scans) are names with a well-known meaning within this world. This convention also holds for the *name=nextzone* attribute.

In the prototype, agents recognise the different names in a world and act correspondingly, for example, to implement a search algorithm. In complex worlds, it may be infeasible to encode all relevant information in a single name attribute, and attribute sets may thus be much more extensive: attributes may for example indicate the type of image stored in a *MultiFileContainer (MFC)* object (e.g., MRI scans or CT scans), or attributes may describe specific properties of the image types (e.g., resolution, pixel depth, etc.) grouped in a specific object. Hyperlink attribute sets may similarly indicate "groups" that a room belongs to; region=EU/VS and scantype=MRI/CT rooms are examples.

In the prototype world, images are of one type only, and there are only two types of file in each MFC, recognised by extension: one containing the raw image data of the MRI scan, and another containing this 3D image's orientation (transformation matrix) (details below). Every MRI scan has a different file name.

The *confined=yes* is a standard attribute for hyperlinks to confined rooms. Agents can request the RoomID of a target room from their MMW, this allows image agents to detect if they have already visited a room before. Note the *name=..* attribute in the confined rooms. *name=..* is a convention which means "back" in Mansion. In the world shown in Fig. 40, the room structure of the world is simple and straightforward, allowing for back links only from confined rooms. The few naming conventions provide sufficient information for the prototype agent to navigate this world.

The following section describes the MRI data analysis application in detail. After that, we describe the agent program used to search the above-outlined world. Subsequently, performance measurements are described for agents that search the (unconfined) rooms, as well as possible optimisations of the Mansion implementation.

10.1.4. MRI data preparation and image analysis

Every hospital in this world provides two rooms: a confined rooms with raw, identifiable MRI scans, (this can be extended to contain more identifiable patient data to allow for more detailed matching, taking contextual information into account), and a regular unconfined (entry) room that contains de-identified MRI scans that can be searched directly. The latter data can be pseudonymised, such that the hospital (but not the agent owner) can re-identify the patient, for example for follow-up questions.

An advantage of searching unconfined data is that the room/data owner need not check the result of a confined agent's search. An agent can take results home directly and only contact the hospital or data owner with follow-up questions if needed. Whether the confined or unconfined approach is chosen depends on the application, and on whether confined search provides agents with significantly more information than a search over de-identified information does—assuming de-identification is possible. The choice need not be binary. It is possible that a pre-preparation agent scouts the de-identified information first, to find out whether there is potential for a more detailed search in the confined room. The confined room, like any room, has an ACL, and before an agent can enter, its owner needs to be authorised; it may be that an agent is allowed into the confined room only after registration, to prevent overloading the room's owner.

A (precompiled) program called *fsl* is used for image analysis, along with supporting programs. Fsl has been specifically developed for (functional) MRI and DTI (a related form of imaging) brain image analysis. To de-identify MRI images, fsl provides a program called BET, for *brain extraction*. A surface modelling approach is used to detect the brain-skull boundary, to remove the skull and facial aspects from the MRI image and keep only the brain.

Assuming that no features inside the brain are in themselves identifiable and that no information about the patient is included in the image header, in the file name or in its attributes (e.g., creation time), it may be sufficient to assume the data anonymous and to allow sharing of the data more or less freely. Entering the world may still require registration as a researcher. BET pre-processed MRI images are stored in a file container object in the zone entrance rooms called *scans-bet* (Fig. 40).

The image processing in our prototype makes use of a few programs that come with fsl. The description of these programs makes clear why the object in the world entrance room contains an object called *master-templates*.

Image processing in the prototype consists of the following steps:

- An agent owner manually draws a *Region of Interest (RoI)* in a *template* image. This allows the image analysis to focus on, to select for example the left frontal lobe. Different *master-template* files for brain-extracted and non-brain-extracted MRI scans are stored in an object in the world entrance room.
- The template brain images must be ‘‘registered’’ with each MRI image in the world; registration means that the 3D-orientation of an MRI image is modified such that it corresponds to the template brain image, so that a RoI can be mapped on it. Registration results in a transformation matrix which is stored in a file separate from the MRI image. Registration is a CPU-intensive task. An advantage of using a worldwide template image, is that hospitals can preprocess their MRI scans by registering them to the template image and creating a transformation matrix before placing MRI images in a room. Each MRI image therefore comes with a transformation matrix that has file name extension *.tr.mat*. RoIs can be drawn in the standard template image available from the world entrance room. A different template is available for brain-extracted scans.
- Given that the transformation matrix is known, it is straightforward to map the region of interest (drawn on the template) over a registered MRI scan, and to do calculations over it. Statistical calculations over the RoI calculate the minimum, maximum, median, and average intensities of pixels in the RoI. A relatively high average intensity compared to an image’s minimum intensity may imply something useful to a researcher, such as whether a potential tumour may be visible inside the RoI; a more realistic agent could implement a more involved algorithm, for example to detect particular contours or shapes (using segmentation), but this is outside the scope of this research.

The image analysis pipeline implemented in our prototype agent consists of the two processing steps described in the last bullet: the mapping of the RoI onto each MRI image using the transformation matrix, to create an intermediate file with everything but the RoI filtered out. This is followed by processing of the intermediate file to calculate some simple statistics.

The two processing steps are executed by bash script, which calls the binaries with appropriate startup arguments. The script is executed by an agent, with as arguments the ‘‘base name’’ of a MRI data file and the transformation matrix. The agent, bash script and fsl programs are executed within the agent’s jail.

10.1.5. A prototype image analysis agent

This section describes a prototype agent. This agent is used for performance measurements described in a later section. The prototype agent simply searches through all rooms for objects called *scans-bet*, downloads their content, and runs the imaging pipeline over one of the MRI scans. Then, it moves on to the next room (which is in the next zone). Since for performance confinement is not relevant, the prototype searches BET-extracted images from the unconfined zone entrance rooms.

After entering the world entrance room, the agent follows all *"name=nextroom"* links until it arrives back in the world entrance room. In every room it visits, the prototype agent retrieves all image and transformation matrix files from the object that contains the (brain extracted) scans and runs the script implementing the image processing pipeline. The script processes the data in the jailing directory, to which intermediate files are written. After executing the script for a given MRI scan, the agent writes the file containing the statistics results into a directory in the jailing directory which, by convention, is written to the AC before migration. In a confined room, this file (export) would be picked up by the guardian agent.

The full prototype agent is shown below (only some comments and details removed):

```
#include <stdio.h>
...
/* C agent libraries */
#include "agtlib.h"
#include "mmw_c_stub_MansionAPI.h"
#include "mmw_c_stub_RMO.h"
#include "mmw_c_stub_MFC.h"

static int err;
static char buf[255];
static char *outbuf=NULL;

/*
 * Execute script (image analysis pipeline).
 */
int do_work(char* filebase, int hop) {
   /* Output is written to a file in agent’s data directory
      is saved in AC automatically by middleware when following
      a hyperlink.
    */
   sprintf(buf, "code/linux_x86/scr.sh %s.nii.gz %s.tr.mat hop_%d_%s",
```

```
                    filebase, filebase, hop, filebase);
   return system(buf);
}

/*
 * Download all content that matches kword (e.g., "scan") from object
 */
int get_and_process_content(int entity_id, char* kword, int hop) {
  ..
  mmw_obj_binding_p ob;
  mmw_mcr_t ob_mcr;
  mfc_dirent de[200];
  mfc_stat s;

  /* Mansion API call returns MCR of forwarding service */
  err = MansionAPI_object_bind(entity_id, &ob_mcr);
  ob = mmw_c_bind(&ob_mcr);
  i = MFC_getdirents(ob, ".", 0, 200, de);
  for (k=0; k<i; k++) {
     if (strstr(de[k].name, &s) {
       MFC_f_stat(ob, de[k].name, &s);
       outbuf = realloc(outbuf, s.size);
       MFC_read(ob, de[k].name, i, s.size, outbuf);
       /* And write outbuf to file.
          File looks something like scan3.nii.gz or scan3.tr.mat
        */
       char *filebase = strchr(de[k].name, '.');
       if (filebase != NULL) *filebase = '\0';
       do_work (de[k].name, hop);
     }
  }
  mmw_c_unbind(ob);
  return 1;
}

int main (int argc, char** argv) {
   ..
   mmw_obj_binding_t *binding = bind_rmo();
   int hop = MansionAPI_get_hopcount();
   mmw_oh_t room;

   /* Initializing libraries with specific arguments */
   mmw_rpc_init(&argc, argv);
   mmw_mos_comm_init(&argc, argv);
   MansionAPI_library_init(mcr);

   MansionAPI_get_current_room_handle(&room);
   /* Check not visited yet, if been here before,
      exit. Write room handle to AC segment [data/visited_rooms]
    */

   /* Obtain list of all objects (by EntityID) from the RMO */
   k = RMO_get_entity_list(binding, OBJECT, 0, 200, buf);

   /* search through objects, do something useful */
```

```
   for (i=0; i<k; i++) {
      /* Obtain AS, search if object matches pattern */
      RMO_get_as(binding, buf[i].entity_id, &as);
      if (strstr(as.as, "scans-bet") == NULL)
         continue;
         get_and_process_files(buf[i].entity_id, kword, hop);
      }
   }

   /*
      Here's the part where the agent programmer should implement
      a migration or cloning strategy. For the prototype, the
      approach is simple: follow all name=nextzone links, that's it.
    */

   /* Request up to 200 hyperlink EntityIDs from RMO */
   k = RMO_get_entity_list (binding, HYPERLINK, 0, 200, buf);
   for (i=0; i<k; i++) {
      RMO_get_as(binding, buf[i].entity_id, &as);
      /* Current AS implementation is a string of attributes
         separated by space characters. Null-termin. is checked.
       */
      if (strstr(as.as, "name=nextzone") != NULL) {
         /* Check if target room hasn't been visited before, check OH
            of hyperlink against [data/visited_rooms] list.
            Also check the target room's zone (compare against
            [data/zones_allowed] list). Details omitted.
            If been there, skip. Else, follow hyperlink.
          */
         MansionAPI_follow_hyperlink(buf[i].entity_id);
      }
   }
   /* If we fall through (no "nextzone"), exit and be morgueified */
   exit(0);
}
```

Fig. 41. Code excerpt from the prototype image search and analysis agent

The agent in Fig. 41 is simple and straightforward. The code consists of a few parts: *main* searches through the room for objects that match a keyword described in its AC; in this case, *name=scans-bet*. For every matching object it finds, it calls the method *get_and_process_files*. This method binds to the object, downloads its content, and runs the image analysis pipeline over it, by calling the script *scr.h* with the appropriate file name. This happens in method *do_work*. The code then searches through the hyperlinks of the room, for a keyword *name=nextzone* and follows that hyperlink.

The agent above assumes a homogeneous world layout, where all hospital rooms can be found using ‘‘nextzone’’ links, and where brain-extracted images are always found in an object in the zone entry room called *scans-bet*, containing the image files with extensions that adhere to the standard naming convention. Few if any attributes are needed to describe entities in this world. The above agent ignores confined rooms, it goes for *nextzone* rooms directly. An agent that searches for confined rooms looks for the *name=scans-raw* or

confined=y attributes and follows this hyperlink, or create a clone to follow it. Worlds where different hospitals use different MRI formats can have attributes describing image formats and parameters in the AS of rooms/objects storing them.

Being able to define simple to use, specific hyperlink layouts for specific applications like the one illustrated above is one of the advantages of using application-specific worlds. As a result, the agent's implementation can be kept straightforward, simply iterating through a set of linearly linked rooms.

An example of what an agent can find in a zone entry room is illustrated below, together with a listing of the files found in a *scans-bet* MFC object. The listing (ls) at the top lists the content of the room as seen by the Mansion administrative shell, *Mash*. Numbers to the left are *EntityIDs*. *O* means Object, *H* means Hyperlink. ''cd'' is a shorthand, which means ''bind or enter''; *cd* to an object (scans-bet) puts the Mansion shell into MFC mode. The ''f'' in the MFC listing means ''file.'' This particular object does not contain directories, but others may, and these would be indicated using a ''d.'' Information about files in an MFC is obtained using the *MFC_f_stat* call, which was also used in the agent code listed above.

```
Mash#: ls
0   H .
1   H scans-raw
2   O scans-bet
3   H nextzone
Mash#: cd scans-bet
scans-bet O5NOYLPD2VTZZ3ZHTXVE3BQ5U74G3LRE_MFC_0_0
Mash#: ls
2008-10-20 18:36:56 f 10573884 scan3.nii.gz
2008-10-20 18:36:56 f      140 scan3.tr.mat
2008-10-20 18:36:56 f 10488424 scan4.nii.gz
2008-10-20 18:36:56 f      141 scan4.tr.mat
2008-10-20 18:36:56 f  9950629 scan5.nii.gz
2008-10-20 18:36:56 f      138 scan5.tr.mat
```

Fig. 42. Example of a room's content and object content viewed using the *Mash* administrative shell

Fig. 42 shows the MRI image files, with extension *.nii.gz* (with an image size of about 10 MB; a single MRI scan from an fMRI time series), and the transformation matrix files (*.tr.mat*) that results from registering the image against the template brain-extracted MRI scan in the object *master-templates* in the world entrance room.

10.1.6. Searching and cloning

If there are a large number of zones (and rooms) in a world, visiting and searching all of them sequentially is often not time-efficient, particularly when there are many rooms. A strategy to speed up search is to create an agent to search each zone onward. Similar applies to any world where a large number of rooms have to be searched, or when these are not linearly

structured and/or if a world supports jumping. If the WER contains links to all zone entrance rooms, implementing parallel search is trivial: just have an agent create a clone (child agent) for each hyperlink from the WER; the parent agent can let each of its children follow a link. The parent agent can remain in the world entrance zone to coordinate work among its children, and/or to collect results. This section describes a cloning agent and its use.

An agent that implements cloning is shown below. It creates clones whenever more than one outgoing link exists that needs to be followed; in the code below, this is checked by iterating over all attribute sets, searching for hyperlinks that are not *name=*. or *name=...* The cloning agent, a variation on the linear search agent described in Fig. 41, has been tested.

If more than one outgoing hyperlink exists, a clone agent is created for each extra hyperlink. Next, the parent waits for a connection from the clone(s) *(MansionAPI_comm_accept*). After the connection is accepted, the parent sends the child a message containing the *EntityID* of the room to enter. Only the parent's code used for cloning is shown. A child checks its clone depth to figure out that it is a clone and contact its parent; other methods exist (i.e., placing information in the AC) exist but are not used in the prototype.

```
h = RMO_get_entity_list(binding, HYPERLINK, 0, 200, buf);

/* Go through all hyperlinks and, if not a self or back link,
 * and if there is more than 1 outbound hyperlink, clone.
 * Text describes subtleties.
 */
for (i=0; i<h; i++) {
   if (RMO_get_as(binding, buf[i].entity_id, &as)) {
     /* filter out self and back links */
     if ((strstr(as.as, "name=." != NULL) ||
         (strstr(as.as, "name=..") != NULL))
            continue;
     r++;
     if (r > 1) { /* Found >2 hyperlinks, clone now! */
        /* Child agent's gaid is put in child.gaid */
        MansionAPI_clone(PURGE_DATA_SEGS, &child);

        /* Wait for child to connect.
         * Child will connect if it finds its clonelevel over 1 */
        connid = MansionAPI_comm_accept(1, BLOCK, &peer_gaid);

        err = MansionAPI_comm_recv(connid, 1024, testbuf, BLOCK);

        /* Send command to do to child, e.g., room to enter. */
        n = strncpy(testbuf, atoi(buf[i].entity_id), max);
        err = MansionAPI_comm_send(connid, n, testbuf, 0);

        MansionAPI_conn_close(connid);
     }
   }
 }
```

Fig. 43. Example code for a cloning agent

The above code fragment is simple; it substitutes the final part of the agent code in Fig. 41. The cloning algorithm works in the world entrance room explained above, but also in other cases where an agent finds itself in a room with multiple outbound hyperlinks. First, the agent searches the list of hyperlinks in the RMO. If there are more than one outbound links (excluding *self (name=.)* or *back (name=..)* links[60]), the agent clones itself so that a child can follow the extra link(s).

After startup, the child agent's code (not shown) will check its clone depth and connect to its parent, which sends the *EntityID* of the hyperlink to follow to its child.

Note that like with UNIX *fork*, the parent and the child's code are the same, except that in Mansion, a child starts from *main*. Thus, the code for handling parent and child functionality should be present in an agent's code (not shown).

An agent can check its clone level using the call *MansionAPI_get_clone_depth*. If its clone depth is nonzero, it contacts the top-level parent whose AgentID is found using the Mansion API call *get_parent(0)*. This suffices if the first agent coordinates all the work in a world, i.e., using a *master-worker* scheme. If clones can also create clones, things may become slightly more complicated; it may then be useful to place (transient) information in an AC segment before calling *clone*, to contain information for the child about who to call and what to do. The AC provides a straightforward and flexible communication mechanism between parent and child that an agent programmer can use.

After connecting, the code of Fig. 43 activates by returning from the *MansionAPI_comm_accept* call. The child sends a question to its (grand)parent, which returns the *EntityID* of the hyperlink to follow.

Some details about the code in Fig. 43. The flag *PURGE_DATA_SEGS* to the *MansionAPI_clone* call is used to control which agent data segments are copied to the child. By default, the child agent inherits all segments of its parent including data segments, such as a visited rooms list. However, a number of segments may not be needed by the child, for example, results collected by the parent. By specifying *PURGE_DATA_SEGS*, no data segments are copied over when the WED creates the clone, except for segments marked using the *MansionAPI_ac_seg_keep_on_cloning* call (not shown).

Keeping data segments is useful when, for example, an agent creates a segment that contains instructions for a child agent on what to do before cloning. Thus, a ''master agent'' may place different instructions in this segment for different child agent (e.g., different keywords for objects or rooms to search for). This mechanism avoids the need to communicate instructions explicitly to every child. Code segments are always kept on cloning; this means that if certain tasks require very different agents, that is, with different code segments, these should be injected as different agents. For more details on the cloning procedure, please refer to Sec. 9.1.7.

[60] The naming convention for self and back-links is chosen for familiarity, as a similar naming convention is used in UNIX directories. Back-links are useful in worlds with a linear or a tree structure, allowing agents to go back to the previous room in worlds where jumping is not allowed (Sec. 2.3.12).

From informal observations, it appears that cloning takes approximately twice regular migration times described in earlier chapters and in Sec. 10.2. This makes sense, given that an AC is shipped to the world entrance daemon, copied there, and sent back. A tradeoff exists: the time it takes to create a clone, compared to the time a task takes without cloning, determines whether it is useful to clone.

A few notes on future optimisations. As described in appendix 6, the clone procedure can be optimised to limit the load on the WED. Such an optimisation is useful: if cloning takes multiple seconds per child, then one agent that creates many clones sequentially could become a bottleneck.

Further, a flag *DONT_MORGUEIFY* could be considered for the clone call. This flag is not implemented, but may be useful to prevent that many ‘‘one-off’’ agents that do a small (sub)task are sent to the morgue, overloading this system and the agent’s owner with returning agents that carry no useful results—particularly if these agents communicate results back to their parent or a master agent. There may be limitations. Sometimes, every sub-agent should return, for example, so the agent’s owner can check each agent’s audit trail for security—e.g., when unsure how many hops a child may take. Having different subtasks return separately may also be more reliable than having a central master (which may fail) collect and return results.

Again, as so often in Mansion, a trade-off must be made on the basis of requirements, scale, and purpose of an application, resulting in a particular organisation of the multiagent application.

10.2. End-to-end performance of the prototype agent

The agent shown in Fig. 41 is used for basic measurements on how this agent performs. These measurements include migration, copying files from the AC, agent startup, copying files to the AC, finalizing the AC, migrating and starting again, retrieving files from an MFC object, running the image analysis pipeline (script + *fsl* analysis programs) in a jail, and migrating again. The *fsl* script execution time is arbitrary; other applications may take longer or shorter to execute. The performance of a given binary program depends on whether a program is system call intensive (chapter 6), and also on whether it requires large (or many) input files: it is impossible to select a program as representative of all types of agents.

The performance measures are made on the DAS3 research cluster[61] which was also used for the tests in Sec. 8.4. In the setup used here, every zone runs on one node, with MOS and MMW on the same machine. All files including the jailing directories of agents, are mounted NFS directories. AOS is the only component that stores its internal files (AC, zip file, segments) locally, on /tmp. Middleware processes were configured to communicate using TCP, without AOS. The tests are end-to-end measurements; micro-benchmarks were described in chapters 4 and 6.

[61] http://www.cs.vu.nl/das3/

The tests are run with an agent that initially, when injected, contains 32 files, with a total size of 34 MB. (The total size of the agent's AC is thus a lot larger than the ones described in Sec. 8.4). The agent, besides the *fsl* programs, also transports two MRI scans of approximately 10 MB each along, as well as a template scan with a region of interest (RoI) drawn in. AC size is arbitrary—an extreme case.

Another mild extreme is in the agent's computational demands: calculating statistics over a single scan takes approximately 60 seconds. Such numbers are not uncommon in (desktop) grids, and Mansion also addresses grid application scenarios (although then run where the data resides). For this example, assume that all hospitals—as most hospitals nowadays do—have a (small) cluster available which allows agents to do such computations, possibly multiple in parallel on different machines. Note that when local computational power does not suffice, hospitals can (dynamically) acquire extra computational power, e.g., from cloud providers, as a ''private grid'' [73].

Note that the task at hand is trivially paralleliseable: each agent (imaging script) takes a single MRI image and analyses it; when more images are available, these can be handled sequentially by a single agent, or more agents can be deployed (e.g., by cloning) to analyse different images in parallel, on different machines. Future work may consider approaches to allow agents to spin off (jailed) subtasks on different machines in parallel, without cloning.

In addition to the performance of the above-sketched task, the base performance overhead of an agent excluding the computationally intensive tasks is measured. These results, combined with results shown in chapter 4 and 6, provide an estimate of end-to-end performance for different types and sizes of agents.

10.2.1. Experimental procedure

Various aspects of the agent's end-to-end time in the world are measured by inserting microsecond timers in both the MMW and agent code. Timings are written to standard output, captured, and analysed later. Agent transfer time is measured, with *finalize* and AC extraction times separate, as well as the overall time that it took from invoking *follow_hyperlink* to agent startup on the next host. The size of the agent's AC is almost identical on each hop; a few small segments are added for the audit trail mechanism and to keep standard-out and standard-error logs of the agent.

The overall execution time differs per host. Execution time is measured as the time from startup to *follow_hyperlink*. On hop2 and hop3, the agent retrieves two and three MRI scans respectively. The time it takes to download the MRI scans to disk (i.e., to the agent's NFS-mounted[62] jailing directory) is measured, and the time to execute the imaging pipeline on an MRI scan (once per hop). As the MFC object runs in an object server located on the same machine as the agent, local RPC calls to the MOS are measured. The MFC object reads the MRI files that are retrieved by an agent from an NFS-mounted file system. The MOS does runs MFC objects as separate processes (chapter 7).

The experiment consists of the following steps. First, 2 measurement series are taken with jailing of agents enabled. Second, 3 measurement series are taken without jailing, to see if jailing influences the various timings during agent execution. Each measurement series consists of 3 hops; averages (and standard deviations) are calculated using the measurements of all hops in the series, so based on measurements of 6 and 9 hops, respectively. Below, the migration times as measured in the experiment are described. Results related to agent execution and jailing measurements are described later in this chapter.

10.2.2. Migration times

The results of the 5 measurements described above is listed below. An agent migrates through the unconfined rooms in the world shown in Fig. 40. The agent is shown in Fig. 41. From the world entrance room (*hop1*), the agent goes to the first hospital zone (*hop2*), from which it migrates to the second hospital zone (*hop3*). In every room, the agent searches for an object names ‘‘scans-bet,’’ and if that exists binds to it, downloads the MRI scans to its jailing directory, and executes the image pipeline script on one of the downloaded MRI scans. If tasks are completed, or if no *scans-bet* object is available, the agent searches for a hyperlink with an attribute *name=nextzone* and follows that.

In the first room the agent does not find a scans-bet object, so its task is done quickly, and migrates onward. On *hop2*, the scans-bet object contains two MRI scans of 10.7M and 11.7M respectively, and two small files containing the transformation matrix, of 140 and 141 bytes, respectively. *Hop3* contains a scans-bet object with tree MRI scans and corresponding transformation matrices, of comparable size (10.6M, 10.5M, 10.0M, and 140, 141 and 138 bytes). *Hop3* does not contain a *name=nextzone* link, so the agent exits and is shipped to the morgue.

Fig. 44 describes agent transfer times, as an average over all measurements.

Measured protocol step (seconds)	Average	Std. dev.
Migration total: follow_hyperlink to startup	6.102	0.186
finalize	2.292	0.041
handoff	1.171	0.016
AC extract code/data segs to jaildir	2.359	0.090

Fig. 44. Timing of total AC migration and its main contributing factors: finalize, handoff, and extraction of code and data segments before starting up the prototype agent. Average over 5 runs, with constant AC size of 34 MB.

Shown are finalize, agent handoff, extraction of all code and data segments to the AC,

[62] Using an NFS mount for jailing directories is suboptimal, but reflects a common use case. Many people use NFS-mounted home directories on shared UNIX systems, and using /tmp for Mansion as a default is not feasible since space on /tmp is often limited. Clean measurements (without NFS) were described in chapters 4 and 6.

and the total time from migration to agent startup which includes the previous three times. All steps measured are identical for each run series on each hop, irrespective of whether jailing is applied or not, so the average over all runs is taken. Standard deviation is also shown.

As can be observed from Fig. 44, measurements are quite consistent, with low standard deviations. The total agent migration time takes approximately 6 second, with most of the time (over two thirds) spent in finalizing and extracting the AC. Consistent with chapter 4, finalizing and extraction of segments contribute significantly to migration cost. The cost of finalizing an AC in AOS is in line with the measurements shown in Fig. 37.

The row *handoff* indicates the time it takes for the Mansion ATP to complete. This is the time needed for shipping the finalized AC, using the procedure described in Sec. 8.3.2. After subtracting finalize, handoff, and AC extraction cost from the total migration cost, only 0.28 second is left, which is the time needed to manage the agent: register an agent table entry, create a jailing directory, creating an agent key and starting a (jailer with an) agent.

AC extraction takes place after handoff, and consists of the MMW extracting all data and code segments into the jailing directory, prior to starting up the agent. This differs from the AC extraction step shown in Fig. 8.4.2, which measures extraction of segments *inside* AOS. The MMW automatically extracts code and data segments to the agent's jailing directory, including the large MRI image files, the agent, image analysis programs (code segments), and data before the agent is started up.[63]

AC extraction times include only extraction of an AC after *follow_hyperlink*. Extraction of the AC in *hop1* right after injection (not shown) is approximately 1.4 seconds aster: the average time it takes to extract the (practically identical) set of code and data segments on the first hop is only 0.953 seconds, with standard deviation 0.090.

10.2.3. Execution times

This section describes the measurements of agent execution times from the measurement series described above, which include measurement of the image analysis tasks that are part of the agents. Two measurement series were made with jailing enabled, and three without (see Sec. 10.2.1).

The agent execution (image analysis) tasks differ per hop. *Hop1* does not have a *scansbet* object available, so here the agent searches the RMO only before moving to the next room (hop). *Hop2* contains an object with two MRI scans. *Hop2* contains three MRI scans. The agent downloads all of them, but runs the imaging pipeline over only one of them (for the purpose of the measurements, the agent was modified to analyse only one image). Different measurement series are run to see if there is a difference in execution times for agents which are jailed versus agents which are not jailed. These are shown in different columns in Fig. 45.

In Fig. 45, every row indicates a stage in the execution of an agent. Every row indicates a different stage or a different hop. On different hops, agents have different things to do. The

[63] Possible optimisations were discussed in Sec 4.7.2.

most relevant results are discussed in order of the rows depicted in Fig. 45.

Row 1 contains the execution time of an agent on *hop1*. All an agent has to do on *hop1*, is search the attributes set of the RMO for an object called *name=scan-bet*. The agent does not find this object, so it then searches for a hyperlink with attribute *name=nextzone*, and follows that hyperlink. Besides searching the RMO, the agent reads and writes a few files from its jailing directory. These are extracted from the agent's AC by the MMW before it is started up (times not included, shown in Fig. 44). These files are created by the agent's owner and contain the keywords (attributes) needed by the agent to find matching objects and hyperlinks. For *hop1* agent execution takes 153 msec if jailed, and 129 msec if unjailed.

Hop / measured task	Jailed	Std.dev.	Unjailed	Std.dev.
1: Hop 1 overall (little to do)	0.153	0.025	0.129	0.010
2: Hop 2 overall agent execution time	60.678	0.216	60.492	0.264
3: Hop 3 overall agent execution time	60.744	0.169	60.215	0.201
4: Hop 2/3 "*.nii.gz" download average	0.497	0.043	0.474	0.056
5: Hop 2/3 "*.tr.mat" download average	816µs	40µs	675µs	22µs
6: Hop 2 image analysis / script	59.047	0.240	58.931	0.189
7: Hop 3 image analysis / script	58.541	0.249	58.177	0.056
8: Hop 2/3 tasks excluding 4–7	0.254	0.002	0.213	0.006
Total (seconds):	239.926	-	238.632	-

Fig. 45. Agent execution breakdown per hop: average + standard deviation. Measurements in seconds unless indicated otherwise. Total adds up all steps 1-8 for jailed and unjailed.

Rows 2 and row 3 show the time of the overall agent execution, including imaging tasks, on *hop1* and *hop3*, respectively. On *hop2*, the agent first downloads 2 MRI scans, and on *hop2* it downloads 3 (different) MRI scans, of nearly identical size. On both hops, it then executes the imaging script once before moving to the following room (or, in case of *hop3*, the morgue). The differences in execution time for the jailed and the unjailed cases of *hop2* and *hop3* is explained in the discussion.

Row 4 and 5 contain the average time it takes the agent to download a large MRI scan of approximately 10 MB and a small file of approximately 140 bytes, respectively. The small file contains a transformation matrix with the same base file name as an MRI scan, but with a different extension (*tr.mat*). It is used to align the 3D of the MRI scan with the template MRI scan carried by the agent, which contains the Region of Interest over which the agent's computations have to take place. The measurements of downloading different file sizes are relevant as these indicate the overhead of invoking objects in a MOS (on the same machine). This way, bandwidth and latency (the overhead of the RPC mechanism for invoking an object) can be obtained.

Rows 6 and 7 show the time the image analysis script (which spawns the several *fsl* tasks needed to analyse an MRI image) takes to complete, in jailed and unjailed cases. Note that

direct comparison between agent execution times on *hop2* and *hop3* (row 2 and 3) is not possible. On *hop3* three MRI scans are downloaded while only two MRI scans are downloaded on *hop2*. At the same time, the size of the analysed MRI scan on *hop2* is larger than the one analysed on *hop3*. As can be observed from row 6/7, the times of the MRI image analysis steps correspondingly differ.

Row 8 shows the overall agent execution time excluding the time needed to download MRI images and run the image analysis script. This time is a base time required by agents to do bare-minimum tasks like reading a few files from the jailing directory and searching an RMO before moving on to the next room. This takes 0.25 and 0.21 seconds, with and without jailing respectively.[64]

10.3. Discussion of measurements

This section discusses the measurements reported in the previous sections. For clarity, we describe migration and agent execution times separately, before discussing end-to-end performance.

10.3.1. Migration times

Given the performance measures in Sec. 8.4, the high AC finalize and extraction costs measured are not very surprising; *zip* and *sync* costs were seen to dominate AC transfer costs in AOS in earlier chapters. An optimised setup, where segments are not *sync*'ed to disk and not zipped as part of finalization but shipped directly (Sec. 4.7.2), may remove some of that overhead, although for larger AC's zipping may be beneficial as it reduces some bandwidth-related overhead compared to sending raw uncompressed segments directly. Possible efficiency gains achievable with certain optimisations may be: a cost closer to pure handoff times (which includes AC transfer) could perhaps be achievable when compression provides little gain, or if compression times are close to the overhead when shipping an uncompressed AC; that may save up to 2 seconds per transfer just on *finalize* in an ideal case, for very large AC's of 35 MB.

The approximately 1.4 seconds difference in AC extraction time between the first and the second/third hops described above, can be explained as follows. After following a hyperlink, the receiving MMW extracts AOS segments into the new jailing directory. On the second and third hop, this happens while the sending MMW is busy cleaning up the old jailing directory at the same time. Both jailing directories are stored on the same NFS-mounted file system on the same LAN.[65] Such concurrent activity does not happen with the first hop, since

[64] The 0.28 seconds reported in the previous section was the average time (over all hops for all measurement series) needed on a hop for the MMW to do bookkeeping to manage the agent and start it up; the times reported here are agent execution times excluding this MMW-internal overhead.

the *inject* program does not remove the input directory with the agent's files after injection. As such, these measurements (0.95 and 2.36 sec) can be taken to present an upper and a lower boundary on AC extraction time, depending on how the MMW is deployed, a difference of 1.4 seconds.

In all, the measurements shown in Fig. 44 for AC migration and handoff represent a ''worst case.'' Speculatively, in an optimal setup where AC's are finalized and extracted using local disks with an optimised *finalize* mechanism in AOS, the transfer of a 35 MB AC could, perhaps, be reduced with between 1.4−3.4 seconds for a 35 MB AC.

10.3.2. Agent execution times

The results provide a rough idea of where the time goes during agent migration and execution. The measurements are quite consistent, also when comparing them to measurements described earlier in this thesis. The relatively small set of measurements (particularly when described separately per hop, as in Fig. 45) leads to relatively large standard deviations[66] in some cases. Nevertheless, the measurements provide a basis to assess end-to-end performance of the prototype Mansion agent running in the prototype world. The observations are discussed below.

Impact of jailing. From Fig. 45 it becomes clear that for most tasks that an agent executes, the difference between jailed and unjailed execution is small, particularly considering other overhead such as the the large cost of finalizing/shipping ACs. For the image analysis task shown on row 6/7, the impact of jailing is compared to the total runtime of the agent (0.116 and 0.364 seconds difference for *hop2* and *hop3*—0.2 and 0.6% respectively). Small overhead is to be expected for a largely CPU-bound image processing task. In absolute terms, the jailing overhead for execution time is small compared to other costs (0.04 seconds difference between jailed and non-jailed agents, rows 1 and 8). Jailing overhead may be larger for I/O intensive tasks than for CPU-bound tasks, as described in chapter 6.

Jailing overhead is noticeable for execution steps that are on the micro- or millisecond timescale. A particular example is row 5, where a small file of approximately 140 bytes (the transformation matrix), is downloaded. Row 8 shows the remaining execution time of the agent after subtracting MRI scan download and processing times for agents on hop 2 and hop 3; this is the time spent on basic tasks comparable to hop 1, where the agent searches the RMO for interesting objects, and reads/writes files to its jailing directory. In both cases, the difference between jailed and unjailed processing *is* significant

It makes sense that jailing is mostly noticeable for execution steps in the order of milliseconds, since the impact of a jailer is typically in the microsecond range (Fig. 25). The

[65] In the prototype setup on DAS3, all jailing directories are mounted on a single NFS-mounted file system.

[66] Standard deviations are relatively large because per hop there are only two measurement points for jailed and three for unjailed agents. Still, measurements are close as can be confirmed by comparing jailed/unjailed averages and standard deviations.

jailer overhead is dependent on the number of system calls made compared to the time spent in user mode; with image analysis programs, most of the time is spent doing computations in user mode, which explains why jailing has had hardly any impact on the times shown in row 6/7. For less CPU-intensive tasks, the relative impact of jailing may be higher, but except for extremely system call intensive applications, other costs such as the migration times reported in Fig. 44 are expected to contribute way more to end-to-end performance of an agent than jailing will ever do.

In conclusion, in general it may be safe to claim that the overhead of jailing is low, particularly compared to other costs observed in Mansion such as the cost of migration or downloading files from an object, although overhead may be larger when an agent executes I/O intensive tasks. This is in line with conclusions drawn in chapter 6.

Object invocation overhead. Rows 4 and 5 of Fig. 45, depicts performance of the Mansion Object Server. Downloading a file of approximately 10M from an MFC object takes approximately 497 millisec in the jailed case, and 474 millisec unjailed. For small files, the difference between jailed and unjailed is noticeable: a download by a jailed agent takes on average 816 microseconds, and unjailed approximately 675 microseconds. Indeed, with such small timings, the performance impact of jailing is expected to be noticeable due to the relative impact of per-system call overhead, at the microsecond timescale (Fig. 25).

Note that in all, 675 µs to retrieve a 140-byte file from an object is not very surprising. In Sec. 4.7, it was observed that a ping RPC call to the AOS kernel takes approximately 127 µs; an object invocation consists of an RPC message from agent to MMW, from MMW to MOS (same machine), from MOS to Object (OII) process, and back. Downloading the same file from a jailed agent takes approximately 816 µsec., significantly more than from a non-jailed agent. Retrieving a file from an MFC also includes the time it takes for the object to read the file from disk, or rather, NFS.

The cost incurred by downloading image files from a Mansion object, about half a second per 10 MB image, is non-negligible. This overhead is for a large part attributed to RPC overhead. If image files were directly accessible from disk (e.g., from the jailing directory or from a shared directory ‘‘mounted’’ by the jailer), this could make a significant difference on end-to-end performance. If Mansion applications use objects mostly for storing (large) files, a file system based approach to replace object access (feasible with jailing—see appendix 9) may be appropriate. If object state is replicated[67], however, the tradeoff may be different.

10.3.3. End to end performance

End-to-end performance of the prototype agent is dominated by AC migration times, MRI-scan download times, and the time it takes to analyse the MRI scans. Given that MRI scans are approximately 10 MB large, and that image analysis is a computationally intensive task, this result surprises little. For comparison, we estimate the end-to-end overhead for a bare-

[67] MFC objects currently read files from NFS internally.

minimum agent, based on measurements presented earlier.

Fig. 44 shows handoff costs of a little over 1 second for a 35 MB agent, and finalize costs a little over 2 seconds. Fig. 37 describes handoff times for small agents: 87 milliseconds to finalize, and 171 milliseconds to migrate a 500 KB AC, and 450 milliseconds and 726 milliseconds for finalizing and migrating a 5 MB AC, respectively. The minimum agent execution cost for a jailed agent observed in Fig. 45 is 0.15–0.25 seconds.

A rough estimate based on the measurements presented in this thesis thus suggests that an agent that is small and does little in each room, may take a minimum of half a second to visit a room and move on to the next one. This corresponds to experiences of the author when running small agents in Mansion, such as the tests reported in chapter 8.4.2.

The obvious result of the measurements presented in this chapter is that using mobile agents is mostly useful for tasks that require server-side computation, or in cases where transferring data to a client computer would pose a large amount of overhead. This tradeoff is visible by the minimum overhead for migrating an agent, which is about half a second in the current implementation, using current-day computers using a fast local area network. Agent transfer can be optimized, for example, by removing zip file generation from the AOS AC transfer protocol, but there is always a basic overhead involved in signing, transferring, checking and starting an agent. It was demonstrated that audit trail verification is very efficient, and that the overhead of the Mansion jailer is, in many cases negligible. Using the evaluation presented in this and earlier chapters, application developers can evaluate whether using agents for a given application makes sense from a performance perspective.

10.4. Discussion and usage experiences

This chapter described the design and implementation of a prototype Mansion world linking hospital rooms so that agents can search MRI data from these hospitals. Each hospital provides one room with ‘‘anonymised’’ brain images, and a confined room that contains raw, identifiable MRI images for search. The confined room may in real life applications also contain additional information about patients.

The main property of the confinement application is that it ensures that *no* information leaves the room except through the room’s owner. In contrast to, for example, e-commerce applications such as discussed in [100], where the result of a distance function that compares images with a ‘‘query’’ image, and possibly thumbnails, are returned to agent owners, the Mansion usage scenario has all results passed to the room owner instead. This example facilitates customized search by agents, while permitting effective evaluation of a researcher’s requests to enlist patients for a trial or for a retrospective studies, using concrete results computed by the researcher’s agents at the hospital side. The researcher (agent owner) sees no results except when a hospital decides to agree to collaborate after evaluating the request and the agent’s search results.

Although the confinement example described as an application is relatively ‘‘heavy-weight,’’ in a context such as medical research the potential gain (e.g., of being able to do research on rare diseases) may be sufficient that confinement is worth the expense of the manual checks and the authorisation steps that may be involved in dealing with customized, agent-supported patient selection requests.

For other applications (e.g., e-commerce worlds containing music or movie stores), automated protocols for exporting selected information, for example in return for payment, may be feasible. Such applications may have less stringent export policies as leakage of a few bits of information is not critical here. The confinement example shows that the concept of a confined room is generic enough to allow for various examples of using agents to search sensitive information, including use cases that deal with extremely sensitive medical data.

This chapter showed that programming agents is straightforward, if the world provides a suitable hyperlink layout. The code fragments shown in Fig. 41 and 43 are real agents, only slightly simplified for readability reasons and reasons of space. The prototype world appears usable as a ‘‘blueprint’’ for worlds for medical research that make information accessible for (confined) search by authorised agents (researchers); by using appropriate attributes, different (linked) hyperlink structures may be created to link rooms of hospitals with different content, if appropriate. It should be noted that for reliability reasons, the "linked link" approach may have to be extended. It would be beneficial to have a links from all rooms to the world entrance room (e.g., marked by an attribute *name=WER*) and from the WER to every zone entrance room. If the world entrance room provides a map, agents can work around zones that are not reachable by going back to the world entrance room and following a link to the next room from there, skipping the failing room if needed. Such considerations are important when designing a world.

Setting up a world is facilitated by a set of administrative tools which allow interactive creation of zones, rooms, objects, and hyperlinks, and inspection of room content as shown in Fig. 42. This works quite effectively: creation of a world requires generation of a world key pair (with a large passphrase), followed by setting up a world zone (again key generation) and the world services; this can be done on a single machine. Generating a regular zone is a similar process, resulting in sending the zone’s ZoneID to the world owner for inclusion in the zone list. When a zone is created, a MMW process and a MOS containing the world entrance room are automatically created. Next, the *Mash* administrative shell can be used to create further rooms, objects and hyperlinks including attribute sets. Overall, Mansion is quite usable from an administrative perspective.

The *Mash* administrative shell can also be used to create agents. It can even be used to communicate with agents using simple typed commands, allowing users to manage interactive agents that execute remotely (assuming no confinement is used). This actually provides great power. The use of interactive agents is outside the scope of this chapter, but a description of Mash is given in appendix 8.

.

Chapter 11
Discussion and Conclusion

This chapter evaluates the design of Mansion and the results presented in this thesis. This evaluation starts from the main research question outlined in chapter 1:

"Can we design a secure distributed system that can be structured such that content can be effectively and flexibly located by mobile agents, which balances security, scalability and controllability concerns of world and content owners on the one hand against the need for autonomy of content owners and agent owners on the other hand?"

This question contains a number of hypotheses:

1. A distributed system can be effectively and flexibly searched by mobile agents if it is sufficiently structured.
2. A world's owner and content owner(s) have security, scalability and controllability concerns that can be met, while the system is distributed.
3. The world/content owner's security and controllability concerns can be balanced against the security and controllability (and autonomy) concerns of content and agent owners that use the system.

Chapter 1 also described high-level design requirements:

- The system should provide conceptual clarity—this relates to the ability to structure a world sufficiently (related to 1—structure);
- The system should be secure in view of possibly hostile agents or hostile/erroneous components of a world (related to 2—security and controllability concerns);
- The system should scale, both in terms of system components and in terms of administrative overhead (this relates to 2—scalability and controllability concerns);

- It should provide autonomy (freedom) for content owners to place content in a world, and for agents to search this content in a way not pre-imposed by, for example, the world designer (related to 3—autonomy concerns).

The introduction phrases the research question differently:

Can we facilitate construction of application-specific distributed worlds, in which users are free to add content but where the system can be sufficiently structured so that it remains understandable by end-users and their agents, thus striking a balance between controllability and security of the system on the one hand and flexibility and autonomy of users to implement their algorithms to find rooms and content—and to process content found—on the other hand?

This phrasing is directed more specifically at the need for a *programming paradigm* that allows for structuring application-specific worlds—and ensuring these remain understandable to agents, so they can effectively find content (or other agents) these worlds.

The underlying assumption (or hypothesis) of this thesis' work is that structuring application-specific worlds is needed and useful; this includes an assumption that *closed* worlds are useful. Furthermore, a hypothesis is that an application-specific structure can be provided in such a way that security and controllability requirements can be met in a distributed setting *and* that effectively navigable worlds can be created. This chapter evaluates whether the above hypotheses hold, and whether the system devised in this thesis, Mansion, indeed addresses the requirements outlined above. We do so by evaluating the effectiveness of the conceptual model as a structuring mechanism, by evaluating the agent migration model (i.e., the *mandatory weak migration* for following a hyperlink), and by evaluating the security and control mechanisms that Mansion provides.

11.1. Structuring worlds—the conceptual model

The Mansion paradigm described in chapter 2 of this thesis provides application developers designers with a conceptual model. This model allows for constructing distributed worlds that, in turn, provide agent (programmers) with a structure to help them find and search information in the world.

The Mansion paradigm centers around the notion of *application specific* worlds. Worlds contain rooms that have hyperlinks between them. For agents, hyperlinks are the primary structuring mechanism. Hyperlinks, as other content, are described using *attribute sets* that allow agents to select appropriate hyperlinks. An example of where this is useful is in cases where a world has various orthogonal hyperlink structures—for example a hospital world which has hyperlinks to rooms containing MRI scans of brains, whereas another hyperlink structure is annotated with attributes indicating rooms that contain CT scans of lungs.

In chapter 10, a prototype world is described which has a very simple hyperlink structure: a linked list of rooms in different hospitals that contain MRI scans. The world has clear structure: each zone (hospital) has an entry room containing anonymised brain scans, and a confined room containing raw brain scans and possibly other identifiable patient information. An agent that needs to search all the MRI rooms in this world can be trivially implemented to follow outgoing links, with relevant attributes, similar to the 2-page C code example shown in chapter 10. This example in itself provides a positive result for the hypothesis that a sufficiently structured distributed system can be effectively searched by mobile agents. This conclusion is rather obvious: if a world has predictable structure, it is easy to search algorithmically.

Are worlds easily designed with a straightforward hyperlink structure? That depends on the application. It is easy to conceive that there is an incentive (medical research) to link up hospital rooms of a certain type to have them be searched by mobile agents because of privacy concerns, if there is a need to have the information searched. Mansion helps to link up hospitals, and having a zone per hospital provides a natural basis for the world's structure. Similarly, structuring a world with auction rooms or a banking world may allow for creating a structure that matches naturally with the inherent structure of the application or the (distributed) parties involved in the application domain.

A World Wide Web-like world where arbitrary content can be added as a Mansion world is much harder to structure: there may be millions of rooms with widely diverging content and, thus, attribute sets—how to construct hyperlink overlays for that? How to ensure each room can be found from an entry room? The search engine-approach taken in the WWW may be inescapable and more natural for such an environment than structuring such a world upfront. (Although one could imagine the search engine to become a kind-of world entrance room, such a world would require links from the world entrance room to be constructed dynamically—a different approach).

When is it worthwhile to construct a Mansion system? It is useful to consider what difficulties may be observed to deploy and use a Mansion world:

- A world designer needs to construct a world structure and to maintain the world's infrastructure (basement) in practice. The application needs to be worth the effort.
- Users must be willing—and have an incentive—to program and use agents to search and find content or other agents in a world. This means dedication to write code for a given task—Web browsers, in contrast, simply display web pages to users without caring about their content; comprehending content is left to users. Once built, a web browser can be used for many applications.
- There may be efficiency drawbacks: a rough estimate (Ch. 10) is that agent migration and startup imposes a minimal overhead of about 0.5 second per followed hyperlink on a fast network. This excludes the cost of running the agent program, which depends on what it needs to do. Compared to visiting many rooms, visiting web sites by retrieving

them may be faster. On the other hand, a well-structured world can ensure the right content can be found more quickly.

- The server-side infrastructure needs to be deployed, and reliably deployed. Although the Mansion middleware is provided as a software package, it must be configured and run: there must be hosts allocated to become a member of a zone and to run the middleware components. The zone must be registered in the zone list, objects (rooms) need to be set up, and sufficient computational power needs to be available to host agents that visit the rooms. This may be more involved and more costly than setting up a web site—although some web sites may be quite involved also.

Before people start building a world, there should be an incentive. In the current world of high-speed networks, ubiquitous mobile devices, Internet/Web access, server-side web services and *Apps* for almost any conceivable task, there must be a reason for going through the trouble of setting up and using a world for mobile agents.

From the reasons for agent mobility outlined in chapter 1, the most important reasons for using agents (and Mansion) are likely **flexibility**—the ability of end-users to customize agent code to an algorithm of their choosing, to allow effective or customized search—and the ability for content owners to **control** the export of information that is sensitive—for example, intellectual property or privacy sensitive information.

Both aspects are addressed in Mansion: agents can be programmed using custom algorithms, even in different programming languages, providing extremely flexible data analysis possibilities to agent programmers. Further, agents are jailed and the confined room construct allows for implementing various ‘‘export policies’’ whose strictness can be determined by a room owner. Finally, Mansion allows agents to meet in rooms to communicate with each other to do business.

Assuming that basic prerequisites for choosing a mobile agent based approach are met, what benefits does Mansion provide over alternative mobile code or mobile agent systems? Perhaps the main selling line for Mansion would be that it helps *‘‘rapidly locate information that needs to be searched at the source*.’’ Mansion’s constructs: application-specific worlds with rooms that can be located using attribute sets and tailor-made hyperlink overlays, allow for annotating rooms such that agents can easily find relevant rooms.

If construction of tailor-made worlds is the selling point for Mansion, it is useful to study how Mansion distinguishes itself from alternative approaches. The following section does that.

11.1.1. Alternative approaches to agent navigation and search

Various approaches may allow for rapidly locate information that can be searched locally at the source. This section explores two example alternatives.

First, consider search engines. What if some data owner simply describes its data set using a set of (high-level) descriptive keywords, much like an attribute set would, and announces the data set on a web page that also provides a method for submitting agents to it? This could indeed be a feasible approach. A disadvantage is that there is no *moderation* of the available data sets or data bases; anyone can announce a data set and advertise it through a Web site and it will be indexed. If the website provides a (standardised) method for submitting agents, remote search of content is possible. But will the content indeed be as promised? Is it of a searchable type? Is there an incentive for content providers to adhere to (application specific) standards for encoding and presenting content so that agents can be easily programmed to search it—such as in the prototype world, where content owners ensure that a standard transformation matrix can be applied to all MRI scans?

If not, agents will have to encode substantial intelligence to cope with the large amount of bogus data or strange data formats that may be found—much like the current Web contains a bewildering amount of content types of varying quality—with some pages containing intentionally misleading content. From an agent (owner's) perspective, it may be quite worthwhile to have application worlds that adhere to certain rules, so that the effort to program agents for it is overseeable, and in which a world owner ensures that content owners adhere to certain pre-set rules and semantics that are found in a world design document and that rooms with misleading content can be removed.

A second disadvantage of using stand-alone Web page like rooms instead of rooms adhering to a world's structuring rules, is that in a Web-like system there is no apparent way to deal with *itinerant* autonomous agents, that is, agents that visit multiple rooms sequentially. The Web assumes an end-user to reside on a client machine from which web pages are viewed. How would an agent travel a world autonomously in search of content, if it is unclear where it will end up when it follows some potentially unreliable information returned by a search engine? How can it be prevented from getting lost? How long will it end up eating resources on the Web, possibly turning rogue down the line? How would authentication and authorisation of agents work in such a system? Is there an infrastructure that allows agents to migrate from web page to web page and take content with them in a controllable way, preferably where modifications to an agent's state can be reliably audited if not prevented?

Given the above—unsolved—questions, it is unlikely that a Web-like system will permit agents to migrate further than one hop at all; it will likely derive to a system for single-hop agents that return to their owner after visiting one page, essentially remote evaluation. This limits the ability of agents to migrate to various systems to search for relevant information on behalf of their owner. Mansion instead provides primitives that support *itinerant* agents, providing structure, (global) resource control, security and agent auditing mechanisms within a single framework.

Another relevant alternative is an agent-centric approach. For example, agent clustering or matchmaking may be used to have agents guide the way toward information in open or closed systems [84]. For open systems, issues such as intentional deceit and lack of standardisation of content may provide similar problems as described above. In a closed system,

however, agents may inform each other of relevant content much like attribute sets do in Mansion. This approach may thus form a usable alternative for Mansion. The main difference lies in that Mansion provides standardized primitives for annotating typed content and hyperlinks using attribute sets, whereas in an agent-centric system, all information required to locate content would have to be provided as part of an agent communication language that all agents understand.

Mansion provides an inherent structure based on hyperlinks that pre-impose structure on migrating agents. An agent-centric approach would require agents to talk to each other to find content. This implies more complexity in the agent. Further, methods for migration would become disjoint from application semantics. In Mansion, hyperlinks are tied to attribute sets. A system where knowledge is present in the agent layer, would require more intelligence and knowledge to be embedded in agents. The disadvantage is that the primitives provided for migration have no relation to the application's semantics; it thus becomes hard to *enforce* structure from the middleware perspective (see also Sec. 2.3.12). Further, hyperlinks are passive and, once generated, static, whereas in an agent-centric solution, agents would have to remain alive for their knowledge to persist.

An important advantage of Mansion's hyperlink structure, is that the primitives provided are simple to understand and use. This means that *if* the attribute sets for given hyperlink topologies are clear, the degree of intelligence that has to be embedded in an agent to find information is limited. Agents that must communicate using an agent communication language to find interesting content need to embed intelligence to ask the right questions and to evaluate the answer before being able to find their way in a world. Although Mansion does provide interagent communication primitives, agents can often do without and simply use a world's hyperlink structure to find information effectively, as shown in the prototype agent.

11.1.2. When to choose Mansion

An assumption is that *closed* worlds have benefits over open systems. The previous section shows how control over aspects of a world such as world topology and content can be beneficial for certain applications. However, it cannot be claimed that structuring worlds for mobile agents is suitable for every application. There must be a use for mobile agents and for the structure provided by hyperlinked rooms, since the overhead to create and deploy worlds for mobile agents will prevent other applications from using Mansion.

For an application where migrating programs to the data is useful, where the data is distributed so that agents must migrate to multiple data sources, where world structure is achievable to help agents to navigate the system effectively, and where allowing agents substantial freedom to search data is useful, Mansion can be a suitable framework to use.

The hospital world described in this thesis substantiates this claim. Chapter 10 presented a simple world for medical research, where zones owned by hospitals are linked in a list-like structure. Rooms contain de-identified MRI data in objects in the zone entrance

rooms, as well as raw, identifiable data in a confined room that is reachable from the zone entrance room. The linked list approach for searching medical data in (linearly linked) different hospitals illustrates that a uniform hyperlink structure helps search.

A production-oriented, large-scale hospital world would likely have multiple rooms per hospital, each for a particular (well-described) type of ward or specialisation in the hospital, and would presumably have links to the world entrance room and from there to all hospitals to ensure reachability, yet it can still have a simple layout. The example linearly linked list (or ring) of hospital rooms is attractive due to its simplicity.

The key point of Mansion is that it *allows for tailoring a world to a given application's requirements, thus allowing agents to execute their task in as efficiently and as simple a way as possible*; a world's hyperlink structure provides the foundation for this.

A critical observation is that Mansion enables a **division of work** between world designers and agent programmers. By having world designers consider suitable world (hyperlink) layouts and corresponding attribute sets, and enforcing these, the amount of intelligence in an agent can decrease. Agents can focus on their task, which is typically to search for suitable content that can be analysed on location using a custom algorithm.

11.2. The design choice to mandate weak migration

A fundamental choice made in Mansion is to mandate weak migration of agents when they follow a hyperlink. Mansion agents consist of static code contained in an agent container. Migration is *forced* whenever a hyperlink is followed.

Weak migration is a given for Mansion's design. In part, this choice was made for pragmatic reasons, since it is hard to support strong migration for legacy applications and different programming languages, particularly in a heterogenous system where few assumptions can be made on the underlying operating system.

Weak migration, however, also simplifies the programming model: every time an agent leaves a room and enters a new one, it is started anew. The programming model is straightforward. Once started, a typical agent checks its AC (e.g., to see whether it is a clone and if so what tasks it has), it checks the RMO for suitable content to search or agents to talk to, and after searching the room, it looks for relevant hyperlinks to follow next, possibly creating a clone if needed to speed up the task. There is little need for agents to remember past actions, other than being able to record results or lists of visited rooms, and possibly information about agents that they have contacted. This information is easy to store in an AC; there is no need for this information to permanently reside in program memory.

Weak migration also makes for consistent semantics when an agent enters and leaves a confined room. At that time, the agent should forget all it has done in the room, which is automatically the case when using a weak migration model.

Finally, weak migration improves mobile agent security. Particularly, it avoids tampering with agent code or data. Agents are always started from the same code and data segment,

from a static (signed) segment in the agent's AC. Middleware processes can check the signature over the initial AC placed by the world entrance daemon, before starting the agent. This ensures that agents are not tampered with on their itinerary, e.g., by some host tampering with their code or data. The only data that can be used to influence an agent's decisions is the data provided to them in rooms or otherwise, and which it may store in its AC. But hosts cannot persistently change the behavior of an agent's program. The static code assumption also allows for implementation of an agent in different languages. For example, as compiled C code for execution on Linux x64 machines, and as Java bytecode for execution on any other machine.

In all, the (weak) migration model is practical and simple. The semantics it provides are clear, and makes programming agents straightforward in general.

11.3. Suitability of the Mansion API

The Mansion API is the interface used by agents to get work done in a world. The API contains several functions: interagent communication, call for finding out about the current context (room, parent agent, etc.), calls to interact with the agent container, and cloning. The functions are not unlike functions provided by UNIX (POSIX) systems, such as communication calls, file system related calls, and process management related calls such as *fork*. In addition, agents are provided with a *binding* call that allows agents to connect to objects, including the RMO, and the *follow_hyperlink* call.

The calls appear effective in achieving their goal. Interagent communication has been used by agents to communicate with their parent agent. Location-independent interagent communication channels allow agents to coordinate multiagent application—for example, synchronising tasks to avoid double work. The RMO, *follow_hyperlink* and binding calls are used intensively to navigate and to search content.

Cloning allows agents to spawn child agents that go to different rooms, for example each following a different hyperlink. Using cloning, multiagent systems can deal with large worlds—although there are bounds on the number of children that an agent can create due to (global) resource protection (chapter 8). As a practical example of using cloning, a master-worker oriented multiagent application was described in chapter 10.

In some worlds, jumping may be allowed—subject to the same migration constraints as hyperlinks. Jumping gives agents more freedom to move, but similar to a spanning hyperlink structure or the web, worlds may become less structured and thus less straightforward to search. When jumping is allowed to compensate for a complex hyperlink structure, searching a world may become less efficient and predictable as a consequence. Jumping seems most useful in very large scale worlds, and is therefore optional, at the world designer's discretion.

11.4. Scalability and controllability

One of the main requirements for the Mansion design is that it is secure and scalable, and that it allows for sufficient control over a world to ensure it adheres to the world's rules. Security and control mechanisms should not interfere too much with content owner and agent owner autonomy or privacy.

Technically, scalability is mainly controlled using the concept of a zone. A world owner controls the world's *zone list*, which is a list of self-certifying identifiers of all zones in a world. When a zone is not in the zone list, it is not in the world. Zone owners are the ''certificate authority'' for their zone, and can sign zone member certificates. This allows for decentral extension of zones with processes, such as object server processes hosting replicas of objects in the zone, or middleware processes on different hosts to manage incoming agents. This way, zone owners can transparently cope with scale increase or decrease. The location independent handles and underlying (distributed) location service are designed to deal with ''elastic'' zone membership changes.

Within a zone, zone owners are free to create new rooms or place content, and agents enter rooms in a zone without the world owner knowing about it. This ensures autonomy of zone owners, content providers and agents in a world. If a breach of world rules is detected—which can be checked by, for example, an agent injected by the world owner—zones can be removed from a world. Decentrally managed zones in combination with the central zone list create the means to balance scalability, controllability and (zone owner) autonomy concerns.

Careful consideration of hyperlink layout is important for overall world scalability. To keep agent itineraries manageable, a world designer may create different hyperlink structures, each annotated with specific attributes. There may be several ''hyperlink overlays'' for different rooms or topics (App. 2), each annotated with different attribute sets, to cater for different types of agents and content. A world designer may prevent zones or rooms, e.g., at the beginning of a hyperlink structure, to experience an unfair share of ''passer-by'' agents by having different world entrance rooms act as different ''entry points'' to a hyperlink structure. This way, adaptation or (manual) intervention is possible when scalability issues arise.

Scalability can be hampered in practice by lack of a sustainable (financial) model to back the expenses required to manage scale increase. The system's distribution model ensures that zones that attract a large number of agents take the load for that interest; other zones of a world do not notice this increased load. The world's entrance zone(s) may also experience an increase of load as a result of a general increase of use of a world (i.e., more agents entering), and should find ways to manage that.

Technically, managing more entering agents entails increasing capacity of the world entrance zone: adding extra hosts, world entrance daemons, (replicated) room monitor objects and middleware processes for hosting agents that enter the world entrance rooms, and additional (replicated) agent location services for managing agents in the world.

In Sec. 3.11, economical aspects related to world deployment are discussed. Fair financial models should be able to support scalability—that is, only the zones responsible for scale increase (due to an increase in interest by content owners and agents) need to scale up, and pay correspondingly. World entrance zones are key to enter a world and must cope with the increase of interest and scale up also. (Note, additional world entrance zones could be added also). This should be payed for, irrespective of what zone in the world attracts the additional interest. Assuming that a world contains content that is sufficiently interesting for users to pay for, business models to sustain a world, its central services, and the world entrance zones as it scales up should be feasible.

11.4.1. Security

A number of techniques are presented for security:

- Host protection is provided through jailing; the Mansion middleware starts up an agent in a jail. The jailer also allows for limiting resource use, to protect against, for example, denial of service attacks, as explained in appendix 3. Agents in a jail cannot interfere with other agents or access files or network resources outside the jail. The middleware can suspend or kill a jailer with an agent's processes in it at any time.
- The Mansion jailer, currently implemented for Linux, makes use of the standard *ptrace* system call, and allows for efficient confinement of binary applications, while avoiding TOCTOU system call argument race conditions that rendered existing user-land jailing systems insecure. Within a jailer, prisoners (agents) can make use of the full UNIX system call interface within their own jailing environment, without being able to damage the system outside the jail.
- Global resource protection is achieved by encoding relevant properties in an agent's AC at world entrance time and delegating some control functionality to middleware processes. At the same time, the world entrance system keeps track of the agent's whereabouts, and of aspects such as the number of children of an agent. Following world constraints, limits can be imposed on the time to live and the cumulative use of world resources by an agent and its children.
- Agents are protected against misuse through an AC integrity verification and audit trail mechanism that, combined with trust in the world entrance system, provides confidence that agents cannot be tampered with without detection. The only time when an agent is most vulnerable is when executing at a host—however, tampered-with agents can be prevented from migrating onwards.
- Self-certifying identifiers are pervasively used as identifiers for world components, effectively by identifying zones. ZoneIDs are used as part of room handles (hyperlinks)

and agent identifiers, which allows for (end-to-end) authentication of processes that host (replicas of) entities by means of the entity's handle. This has the important implication that, from a security perspective, the location service need not be relied upon to provide correct information. If a handle—e.g., of a hyperlink that is registered in a RMO— is trusted, the connection to the entity can be authenticated.

- Every object, including room monitor objects, have an access control lists based on self-certifying identifiers (AgentOwnerIDs). This allows objects and rooms to determine whether agents may access the object, or room. AgentOwnerIDs may also indicate a world-specific *role* or agent (payment) type; the system is flexible even though it uses a simple, short identifier and an ACL mechanism for access control.

Since a Mansion world is closed, it also makes for a closed security model—the security architecture is published as a whole in [79]—that covers most if not all aspects of mobile agent security: controlling world coherence; secure naming of distributed zones, addition and removal of zone (members) by managing zone member certificates; managing agent identification and secure agent location updates; agent owner identity or role-based access control for objects and rooms (together with a confinement system discussed in the following section); accounting for, as well as controlling global and local agent resource usage; protection of hosts against malicious or resource-consuming agents and against hostile object code in an object server; and protection of agents against tampering on their itinerary by preventing illegitimate modifications to an agent's code or other persistent AC segments from spreading beyond their current host.

The security architecture cannot protect agents against hostile hosts directly, although the handoff protocol allows for detection of malicious AC modifications, and some techniques (e.g., code obfuscation) exist that may help agent owners protect their agents in practice. A host controls all processes it manages, so it is hard if not impossible to protect agents against their host; protection mechanisms should be devised at another layer of abstraction, for example using reputation-based mechanisms. The Mansion security model provides a basis for these and for ensuring misuse can have consequences, at the very least in the form of being banned from a world.

11.5. Confinement

Mansion provides *confinement* as a tool for protecting sensitive information which permits agents to search this information *on site*.

Chapter 10 describes an application where a trial or retrospective medical study can be prepared by agents that search highly privacy-sensitive information in confined rooms of different hospitals. Here, agents search information while being unable to export any information; all communication channels are suspended, the AC is blocked and—as always—the

agent is restarted when exiting the room[68]. After finding relevant information, agents can write the results to an *export file* in their directory, which is picked up by a Guardian Agent when the agent leaves. The results are passed to a doctor who can evaluate the results, the agent's credentials, and possibly an earlier a separately submitted proposal to participate in a trial or study, to evaluate if the agent's owner should be contacted to discuss the results or the proposal. Whether this happens is up to the visited hospital; no information is sent to the agent (or its owner) directly.

The above application is extremely strict, as no information may leave the hospital—not even a single bit—without checking. For medical information, this may be essential as information leakage can have critical side-effects; for example, one bit may be sufficient to indicate whether the prime minister admitted to the hospital's neurology ward last week has a brain tumour. Because setting up trials or (retrospective) medical research is costly, difficult and time-consuming, particularly for rare diseases, using confined agents may be worthwhile to alleviate these issues.

In contrast to existing approaches that are typically based on requesting hospitals to find patients manually, the approach described here allows for requesting participation based on the results of a prior *targeted* (confined) search over raw patient information, of which the results are only known to the hospital. The data controller (the hospital) stays in control over whether any information about the search results will be disclosed, to whom, and on what terms. The confinement approach improves the chance that (rare) patients that have a particular disease can be found and—in collaboration with the hospital—can be asked to join in, for example, a clinical trial.

The confinement application for hospitals may be a *killer app* for mobile agents, since it is hard to see any other fully privacy-preserving approach that allows for finding patients in (hard to indexed or, often, wrongly annotated) raw data without risking information leakage using end-user customiseable search.

There are useful confinement scenarios where the risk of leaking one or a few bits of information is less critical than with medical information. For example, for commercial information, occasional loss of data may be less of a problem than with legally/ethically sensitive medical information. Here, use of automated mechanisms to facilitate export of information from a confined room directly to an agent or the agent's owner—for example, in return for payment—may be feasible. As an example of such a scenario, an agent may pass a list of identifiers for content that an it found interesting to the Guardian Agent. The guardian agent may then prepare a web page from which this content can be downloaded, and contact the agent after it left the room and pass the web page's URL to it[69]. Next, the agent or its owner can visit the web page, pay, and download the content.

[68] Covert channels can be prevented as agent execution takes place completely within the trusted hospital environment; agents should not be allowed to determine their own time of leaving. Either a fixed return time should be determined up front, or agents should be killed on exit without being returned. The latter agents should be *clones* in the chapter 10 world.

[69] Identifiers are used for simplicity and to avoid steganography that hides information in data; the guardian agent can find the agent's AgentID in the export file it left in the confined room.

Several other applications exist for searching (less sensitive) data, for example in the scientific domain—having agents of art historians search for particular types of dogs in high-resolution scans of paintings, is just one example. Earlier research explored similar examples for searching commercial intellectual property remotely [22,100].

Other examples of searching highly sensitive material in situations where content owner control is critical, exist. For example: finding relevant case files in police departments may be hard to do securely (and legally) without confinement. In such cases, strict confinement may be a solution.

11.6. Performance

This thesis described performance measurements for most critical components and protocols of the framework, as well as end-to-end measurements.

The jailing system has been shown to support execution of arbitrary binary programs, shell scripts and interpreted java code in a JVM using a standard jailing policy. Overhead of jailing is often negligible compared to other cost such as the overhead for migration; only with I/O intensive agents may the overhead be significant compared to running the agent without a jail. Jailing or sandboxing is imperative for host protection. The jailer supports multiple agent programming languages—including agents containing legacy compiled code—and it provides a way for resource protection. This makes the jailer an effective and secure alternative to language-based sandboxing.

Migration, agent container related operations and the overhead of retrieving data from objects are not inhibitive, but can be optimised. ACs are zipped, which causes overhead that can be removed by an implementation without zip files. For objects, content may be replicated so that agents can read data from a local object replica, decreasing latency. In the prototype, agents download data from objects to their jailing directory to interact with it there; other agents may interact directly with objects using method invocations. If (fast) file interactions are important for an application, a future version of Mansion may support transparent mounting of (object) files in the agent's jailing directory (see appendix 9).

It is clear that world design has an impact on end-to-end performance: the overall time it takes an agent to complete a task depends on an agent's size, the number of hyperlinks it follows—given at least 0.5 seconds overhead per migration—and the time an agent spends inspecting data in different rooms. When agents spend a long time analysing data in a room—arguably, an important argument to use agents is that they can autonomously inspect (large) data sets at the source—without being overly I/O bound, the overhead of jailing and agent migration becomes negligible compared to the overall costs, as seen in the prototype application. As with many other aspects of Mansion, whether the overhead is acceptable from an end-user perspective depends on the application.

11.7. Overall conclusion

Chapter 10 demonstrates how Mansion can be used to develop efficiently searcheable mobile agent applications. The main research question:

"Can we design a secure distributed system that can be structured such that content can be effectively and flexibly located by mobile agents, which balances security, scalability and controllability concerns of world and content owners on the one hand against the need for autonomy of content owners and agent owners on the other hand?"

can be answered positively.

We have shown that it is possible to design a distributed system that can be effectively searched by mobile agents; the world's structure, enforced through its hyperlink layout, has shown to be effective in guiding agents through a world, with simple hyperlink layouts decreasing the effort needed to program the agents. The 2-page prototype agent shows the use of dividing the work between world and agent developers—i.e., of having world designers pre-structure the application for agent developers.

We have shown that Mansion addresses security problems, scalability issues, and the difficult issue of integration of (binary) legacy applications that existing mobile agent systems do not handle. The problem of integrating legacy applications is solved by using a jailing system. Using the jailer as a starting point, we also showed an application that provides a strong case for using mobile agents. Searching hospital rooms with privacy sensitive information where not even a single bit of information may leak out without the data owner's explicit permission can likely only be done using the confinement approach presented in this thesis.

As described in chapter 1, the tradeoff between controllability and autonomy was an important and recurring theme while designing Mansion. Agents (and their owners) are autonomous in that they can implement programs and algorithms of their own choosing, using a programming language that best suits their needs. Except for being sent to the Morgue when agents run out of their allocated resources or time to live or (otherwise) harm the system by their current host, no entity in a world can force actions upon an agent: they are autonomous. An agent owner can select a world entrance system—assuming a set of world entrance rooms is present—that contains the agent location service and world entrance daemons it trusts; this zone contains the components for tracking and managing agents and their resource use, and ensure that modifications to an agent's AC are properly audited while preventing attacks such as audit trail rollback.

To facilitate scalable world deployment, a hierarchical world administration framework is developed. The world owner is in charge of which zones are admitted to a world, on what terms. Zones are managed autonomously, allowing zone owners to extend their zones with middleware processes to handle increasing load as needed. This provides the basis for dealing with scalability issues from a technical as well as an administrative perspective. The world owner does not have to control over every detail of a world, which would interfere with zone,

content or agent (owner) autonomy or privacy. From a scalability perspective, the Mansion architecture attempts to provide fairness in terms of load distribution, avoiding that increased load on one zone does not adversely affect other, less popular zones in a world. This balances controllability and autonomy.

A security architecture is provided that covers most, if not all, aspects of mobile agent system security. Hosts and agents are protected using jailing and the handoff protocol. Zone-wide or per-room policies for room entrance and per-object access control are possible. Self-certifying identifiers ensure that handles can point to (distributed) entities that can be authenticated directly, end-to-end, without having to rely on a central trusted component such as a trusted location service. If needed, certain components—such as the agent location service—can help enforce world rules (e.g., by checking whether migrations between zones are allowed), but most checks are made decentrally by the middleware processes that guard the interactions (API calls) of an agent with the outside world.

The aim of Mansion as a programming paradigm is to provide conceptual clarity to agent programmers, administrators and other users of the system. Mansion ensures that for a given world, core aspects remain predictable—for example, the world's layout, object interfaces and attribute sets, which are defined in the world design document. In contrast to other mobile agent systems and many other distributed systems, Mansion works best when (distributed) applications lend themselves to structuring; it is specifically aimed at designing closed worlds for specific applications. Using a suitable world design, room and content owners know precisely what to expect in a world, so that agents can be straightforwardly programmed to search the world, interact with other agents, and find relevant information there. The prototype world shows the approach is feasible.

Using a framework like Mansion holds promise for constructing future distributed mobile agent applications and systems. It can provide a closed, scalable, and manageable infrastructure which provides sufficient structuring mechanisms to make worlds understandable to content providers and mobile agents, while at the same time providing sufficient room for decentralized management, including the possibility to autonomously link in rooms and content of different owners in separate independent zones in the system—potentially at very large scale.

The Mansion paradigm provides the mechanisms needed to control what content is visible and reachable to agents in a world. It uses simple, clear concepts like rooms, hyperlinks, objects and agents, identified using location-independent handles. Hyperlinks are managed decentrally by room owners, subject to central world design rules, thus allowing for scalable, flexible, controllable distributed systems where content is reachable and can be found in different, application-dependent ways.

Our experiences with the prototype world show that the Mansion paradigm not only works—it works well.

.

Chapter 12
Summary

Mobile agents were first conceived in the 1990s. Since then, various mobile agent systems have been designed, each focused on some relevant part in the design space of mobile agents: secure execution of mobile, portable (interpreted) code, interfaces for interacting with the environment or with other agents, directory services for locating agents, or on applications that require disconnected use or that use multiple agents to complete a task.

Most existing systems impose few constraints on the environment, for example on how agents may join the system or where they can migrate to. Mansion changes this: Mansion focuses on a programming paradigm for mobile agents which is controllable. Agents are mobile, but cannot migrate anywhere: they must adhere to certain rules conceived by an application designer. Within the rules of the application, agents are autonomous: they can be written in any programming language and contain any algorithm to search for information or to locate and interact with other agents, but they cannot escape the confines of the application.

The environment that Mansion provides to agents is called a world. A world is implemented as a distributed system consisting of various rooms that can be reached by following hyperlinks. The hyperlink structure originates from a world entrance room. This environment has similarities to the Web in that it uses hyperlinks, but it is also very different: it is impossible to enter a room without entering a world entrance room and following hyperlinks to migrate to the room first. Further, to be able to migrate to a room, an agent first has to physically migrate to a machine that hosts the room and that is trusted by the room's owner. Because agents run on a machine controlled (or at least trusted) by the room's owner, it becomes possible to enforce security policies on agents and on the way in which agents navigate a world and on how information flows from a room to the outside world.

Worlds are application-specific: their content and structure depends on the application, and the hyperlinked structure of a world ensures that some structure can be imposed on it, to help agents find their way and to locate content efficiently. Various types of world can be constructed, from very large-scale, loosely structured worlds to highly constrained, relatively

small worlds for specific applications.

A prototype world has been constructed consisting of about 20 rooms, with each 2 rooms representing a hospital, which are hyperlinked as a linked list of rooms. An agent can enter the world and seach for sensitive content by entering the first room of the world and searching it, then moving onward to the next room. To search sensitive content, a room is provided by each hospital that *confines* the agent. In the prototype world, these confined rooms contain (personally identifyable) MRI scans.

Confined agents can search content, but when they leave the room they retain no memory of what they found or did in the room. An agent must write any interesting content that matches its search parameters in a list and pass it to a *guardian agent* that is provided by the confined room. After inspection of the list, the guardian agent may pass information on the items in the list to the room's owner. This room owner can decide to contact the agent's owner. This example is, for example, useful when searching for patients with a rare disease in preparation of a clinical trial—the important factors being that the agent cannot leak any sensitive data to the outside world and that there is sufficient incentive to find patients by searching datasets of sensitive data spread all over the world.

Other, less sensitive applications of Mansion can be more automated. An example is an agent searching for movies or music files, where the guardian agent may simply take the export list to prepare a shopping cart using which the agent's owner can later buy the content found by its agent. Agents can also meet and interact to speed up their search or to negotiate a deal. The main function of rooms—with or without confinement—is to provide content that is difficult to index or find using a search engine, and to allow this content to be accessed and searched directly, *on-site*, by mobile agents.

Content in a room can be stored in an object: objects are passive and have methods that can be invoked by agents. Objects can only be invoked by an agent in the same room. Agents can help other agents to find information: agents can communicate with other agents of which they know the *agent identifier*, using migration transparent communication channels. Agents and objects are registered in a special object in the room, the *Room Monitor Object (RMO)*, which also contains hyperlinks to other rooms. Each entity (agent, object, hyperlink) is annotated using an attribute set that allows agents to locate content effectively. The world designer determines what attributes are allowed in a world, based on application requirements.

The system is designed for scalability and security. Rooms and objects can be distributed over multiple physical machines, to allow the system to scale when many agents enter a room or access an object. Each room, object and agent is known by a location-independent, self-certifying identifier that not only functions to locate the entity, but also allows for authenticating it—the entity must be able to authenticate itself using a public key that corresponds to the self-certifying identifier. Each object (including the room monitor object) is accessible only from a middleware process that is a member of a *zone*: a zone is a set of processes indicated by a single self-certifying identifier that indicates the certificate authority of this zone. The middleware processes hold the same self-certifying zone identifier as the objects which are accessible from these middleware processes. The identifier of every zone in a world is

registered in a world-wide service. This per-world service is managed by the world owner and ensures that only trusted zones are part of a world. This bootstraps a semihierarchical trust model of a world.

In addition to the world's zone structure, a (nonhierarchical) structure of hyperlinks between rooms exist. Each room owner can, independently from other room owners, add hyperlinks to other rooms to its room. In some cases—such as the medical world described above—a world owner may impose hyperlink constraints on a world that limit how rooms may hyperlink to other rooms. Other worlds may be loosely structured, where rooms may link to other rooms in the world as their owners see fit.

Other important components of Mansion are the world entrance rooms, through which agents can enter a world. In a world entrance room, agents can find information about the world and follow hyperlinks to enter the (first) rooms of a world. Thus, an agent can be helped to find its way. A world design document describes the main constraints of a world, so that agent programmers can ensure their agents can locate content or other agents effectively.

For security, the system presented in this thesis provides several mechanisms. A jailing system allows for protecting hosts against malicious agents, and for confining agents effectively, irrespective of the language in which the agent is written: the jailer confines binary Linux processes (which may contain an interpreter) such that these processes can only access a scratch jailing directory and can only connect to specific, predefined network endpoints. Starting from this, the Mansion middleware can ensure that an agent can only invoke (allowed) methods of the Mansion API. The Mansion API contains the calls to follow hyperlinks, invoke objects and connect to agents, and enforces both logical and security constraints on what an agent is allowed to do. The middleware is RPC-structured: it consists of several processes that invoke each other using RPC calls; agents are also executed as independent processes that invoke the Mansion API using RPC.

Besides the jailer, the middleware system makes use of an object server (containing the Mansion objects, currently written in C++), and it uses the Agent Operating System (AOS) which provides primitives for storing agents in so-called agent containers (ACs), and for shipping an agent in such a container when it migrates. An AC contains segments with the agent's code, initial data and instructions, as well as data collected on its way. When an agent migrates, hashes and properties of its segments are contained in a data structure that is iteratively signed by the sending Middleware processes, thus constructing an efficient *audit trail* that ensures that tampering with an AC can be detected by the agent's owner. By coupling audit trail verification to a handoff protocol that updates the agent's address in one of the world's agent location services, a tampered-with agent can be contained at its current location, protecting (distributed) resources at the world level against rogue agents at the same time.

All combined, Mansion provides a structured, closed environment in which the most important properties can be controlled by the world owner, yet where agents can implement their own algorithms for finding content and for searching (raw) data in the world autonomously for their owners.

A prototype world shows that it is possible to design a coherent world structure for which programming agents is straightforward. The prototype agents are small, yet they can navigate the world and search its content autonomously in an efficient way. The underlying trust model and security primitives ensure the system's security and scalability. An exploration and implementation of applications is described as validation of the concepts, the design, and the implementation presented in this dissertation.

Appendix 1
The world design document

Below document gives a concrete example of a WDD, taken from the current Mansion implementation.

```
worldinfo {
    owner=YOUREMAILADDRESS,
    scid=THISWORLDSCID,
    name=THISWORLDALIAS // As in the Mansion Nameserver.
}

wddparms {
    // Set this to 1 if you require an administrator email address to be
    // specified when a zone is registered. 0 means that ''anonymous''
    // zones are allowed (but note that the world administrator always
    // requires your email address for verification, in general).
    ZONE_ADMIN_CONTACT_MANDATORY=REQUIRE_CONTACT

    ALLOW_JUMPING=0 // or 1 if jumping is allowed.
}

 // Allowed agent types.
agent_binary_subtypes {
   linux_x86
}

 // This list limits the allowed object types in a world.
objectclassnames {
     MansionObject,
     RMO,
     FileContainer,
     MFC,
     OwnedMFC // each agent has its own ''private'' directory in this object.
}

 // Here, non-standard attributes in the attribute sets (ASes) of agents,
 // objects, and hyperlinks are specified. Note that some fields, such
 // as EntityID or EntityType are not specified here; these are mandatory
 // by Mansion design/implementation and cannot be overruled or changed here.
template_agentAS {
   <optional> name=
   <optional> description=
}

template_objectAS {
   <mandatory> ObjectType=<MansionObject|RMO|FileContainer|MFC|OwnedMFC>
```

```
   <mandatory> name=
   <optional> ContentDescription=
}

 // Note: unique applies to intra-room uniqueness of the attribute only.
 // Global uniqueness can never be enforced.
 // Having a local unique name for hyperlinks is useful to construct
 // globally unique, human-readable ''pathnames'' starting at a given world
 // entrance room.
template_hyperlinkAS {
   <mandatory_unique> name=
   <optional> description=
}

 // Some security stuff. Defines the cryptographic strength of (SSL) connections
 // over which all communication is routed in Mansion.
 // Comparable to the OpenSSL library initialisation strings.
ciphersuites {
   // Currently only RSA authenticated key exchange with AES 256 cipher block
   // chaining crypto, and SHA (1) message authentication.
   SSL_RSA_WITH_AES_256_CBC_SHA
}

zoneinfotemplate {
  // In this particular zone description template, any information (free form)
  // can be filled in - particularly a name - or even nothing.
  // If the world owner acts more like a CA for the world's zone list, it is
  // likely that more required (mandatory) fields are defined, and checked at
  // registration time by the world owner.
  <optional> name=
  // zone administrator's mail address
  <optional> contact=
}
```

Appendix 2
A hyperlink constraint language

This appendix describes a hyperlink constraint language (HCL).

The world design document can contain a HCL, to instruct content providers and users of the system on the way in which rooms may place hyperlinks to each other. Ideally, a HCL should be enforced by the middleware, to ensure that agents cannot migrate through hyperlinks that are not allowed by the HCL. However, actual enforcement of migration rules in a distributed system is difficult. Some practical notes and solutions are proposed this section. At least, a HCL allows world designers to communicate hyperlink constraints to developers and users of the system. This section describes a possible HCL design. It has not been implemented.

The content of a hyperlink constraint document (HCD), based on an informal example HCL, is given below. A key notion in this HCL is *allow_from*. *Allow_from* specifies from which zone to what other zone links are allowed.

```
// It is possible to define a named set of zones. Zones can either be
// ZoneIDs or the zone names that are registered with the world in
// Zone Descriptions which can be found in the world's zone list
set A { zone1, zone2, zone3 }
set B { zone4, zone5 }
// One can also use zone names directly; WEZ is a commonly used
// name for the world entrance zone.
allow_from WEZ to A
// B may link to the WEZ
allow_from B to WEZ
// A can link to B (back is not allowed)
allow_from A to B
// we can create exceptions too: zone3 may not link to B.
deny_from zone3 to B
// but it may link to zone 4; another exception
allow_from zone3 to zone4
// ... And anything not explicitly specified above, is not allowed.
```

The above is a HCD which constrains how agents may ‘‘move’’ in the world. Here, the WEZ can link to any (room in any) zone in set A, the zones in set A (except for zone3) can link to any of the set of zones in B, and B can link back to the WEZ. Like Fig. 40, a circular structure becomes visible. In general, the WEZ would be allowed to link to any room; the WEZ

will often be owned or managed by the world owner, and would be trusted to manage links properly. However, in cases where multiple world entrance zones exist, these may cater specific subsections of the world. Such sections can correspond to the sets above, and corresponding constraints may be useful.

To describe the hyperlink layout of fig. 40, a HCD could look something like this:

```
// assume this world consists of zones hosp1, hosp2, hosp3, ..., hospN
set A { hosp1, hosp2, hosp3, ... hospN }
allow_from A to WEZ // all zones may link back to WEZ
allow_from WEZ to hosp1
allow_from hosp1 to hosp2
allow_from hosp2 to hosp3
allow_from hosp3 to hosp4
  ...
allow_from hospN-1 to hospN
```

The above defines the hyperlink layout in Fig. 40: all hospital zones may link to the world entrance zone, and there are pairs of zones between which unidirectional hyperlinks may exist to form a list or a ring. Note the HCD's restrictions: linking from WEZ to all hospital zones is not allowed. For that, *allow_from WEZ to A* should be defined.

Note that the language as defined above is static. When a new zone is added to the world, we need to change the HCD (possibly, as part of the WDD) as a whole. In particular, the *allow_from WEZ to hosp1* should be changed such that the WEZ may have a hyperlink that points to the new zone, and the new zone should link to *hosp1*. This is inconvenient. A possible improvement may be that a world designer can define names for groupings externally, for example, in the world's zone list.

An alternative to a static per-world HCD may be to have the world owner specify constraints slightly less formally, for example, on a Web page that describes the zone registration procedure and the general layout of the world in human-readable terms. This can then result in *local*, per-zone access control policies that determine what zones agents may be allowed to migrate from to a particular zone. Note that, as described in Sec. 2.3.12, it is irrelevant whether the incoming agent arrives via a hyperlink or via a jump call; hyperlink constraints constrain migration.

As a concrete example of how to implement a decentralised approach, zone-based hyperlink/migration constraints can be embedded in the zone description of each zone. Thus, instead of having a global HCD, zone owners may define from what zones they accept incoming hyperlinks/agents. Note that at the time that zone descriptions are loaded into the zone list, the world owner may verify that their *inbound* hyperlink constraints are not in conflict with the world's general hyperlink constraints.

Zone descriptions provides additional flexibility. For example, zone owners may describe what zone entrance rooms a given zone has, and then from what zones agents may migrate to what zone entrance rooms. A zone may have one ZER for agents coming from zone A and zone B, and another for agents coming from zone C. If deployment-time checks

are possible maintaining decentral policies is a possibly more flexible way to implement hyperlink constraints than to define a formal, static, world-maintained HCD.

So far, fine-grained policies on hyperlinks at room granularity have not been proposed. Instead, the focus has been on constraining migration based on zone information. This has three reasons:

1) Creating global hyperlink constraints would be even more difficult with constraints at the room level, since there may be many more rooms than zones.
2) A room should not know the previous room, visited before entering the room. This can give away a lot of information about the agent (owner's) interests, which is undesirable from a privacy perspective.
3) It would be hard to enforce policies at the room level, as it is not possible to authenticate a room from which an agent arrives; only zones can be authenticated (using the zone-based authentication protocol underlying agent handoff). Zones are the unit of trust in Mansion: at best, the sending room can be known if the sending zone includes this information in the handoff protocol, if the receiving zone trusts the information that the sending zone provides. For both privacy reasons and for simplicity, such a model was decided against.

Overall, the most scalable approach is considered to be to define general hyperlink constraints at the world, using a central HCD to guide rules in a general sense, and to define detailed (inbound) migration constraints in the zone description. An example zone description policy: *allow_from A to Room1 ; allow_from B to Room2*. Such a constraint *would* be enforceable, as each MMW authenticates the incoming agent's MMW using zone-based authentication. This approach does not reveal what the agent's interests are, as it is not visible to what room(s) an agent went in the previous zone.

As a final example, consider the following approach. A world designer conceives a global hyperlink topology containing three sets of zones called sections. One section consists of zones offering travel packages, the second section consists of zones offering houses as second homes, and a third sections offers cars for rent or for sale. Agents travel from one section to another only through the section's entry rooms: an agent in a room for cars may not directly jump or follow a hyperlink to a room for houses. Can we structure this world using zone-based hyperlink constraints, providing sufficient freedom to the different sections to manage their own room topology internally while having sufficient overall control over how agents can migrate in this world? An example topology of this world is shown in Fig. 46.

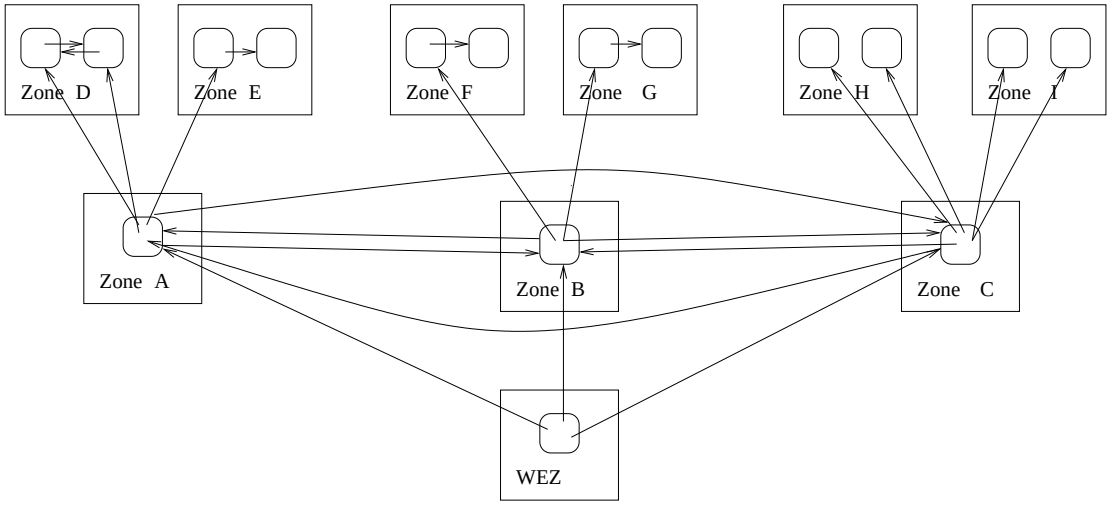

Fig. 46. An example of structuring a world using zone-based hyperlink constraints.

The above-shown world is translated to the following world constraints. To solve some of the problems posed by not having global control at the room level, a set of intermediate zones (zone A, B and C) are introduced in the world as entry points to the different sections.

A hybrid zone-based policy is defined using a global *HCD* combined with *inbound* hyperlink constraints that are defined in zone descriptions.

```
set intermediate_zones { zone A, zone B, zone C }
set travel { zone D, zone E }
set second_homes { zone F, zone G }
set cars { zone H, zone I }

// the world entrance zone links to A, B and C
allow_from WEZ to intermediate_zones

// and A, B and C link to the different sections
allow_from A to travel
allow_from B to second_homes
allow_from C to cars

// Intermediate zones can link to each other and back.
// Note that a set is only a notational aid. A set does
// not imply that zones in it may link to each other.
// Below rules ensure intermediate_zones can.
allow_from A to B; allow_from B to A
allow_from B to C; allow_from C to B
allow_from C to A; allow_from A to C
```

The above zone-based hyperlink constraints ensure that hyperlinks between the different sections *travel*, *second_homes* and *cars* are not possible. Specific zones (A, B, C) function as ‘‘hubs’’ to different subsections. Alternatively, a constraint *allow_from WEZ to all_zones* could be defined. However, then the work of keeping links to various rooms in the world

would be on the world entrance room administrator. Instead, in the above policy, different zones (A, B, C) are responsible for keeping track of their subsection. The HCL formalises that the different subsections cannot link to each other—they can only be reached through the intermediate zones: *travel, cars* and *second_homes* sections cannot link to each other.

What the above HCD does not do, is define fine-grained constraints of exactly to which rooms in a particular zone, another zone may link. In Fig. 46, the zones in *travel, second_homes* and *cars* can each define constraints (i.e., zone entrance rooms) in their zone description. For example, zone D and zone I each have two zone entry rooms, while zone E, F, G and H each have a single entry room. Instead of defining constraints in the zone description, zone F can also simply tell the administrator of room B that it may only link to this specific room; in fact, a convention is that the first-created room of a zone (with a RoomID ending with *RMO_0_0*) is the default entry room, and this is the room that would normally be linked to, so in many cases this need not be told explicitly[70].

The HCD policy above partitions the world in subsections (*travel, cars, second_homes*) which may not link to each other. This particular policy cannot be enforced in a hard way by the world designer (since agents migrate from zone to zone without involving the world administrator). Thus, the world owner must trust the zones in the world to adhere to its HCD. As said before, if a zone violates the HCD, the zone administrator has the power to remove the zone from the world by removing the zone from the zone list.

Note that another conceivable solution for hyperlink constraint enforcement, is to have the ALS enforce the HCD. Since the ALS is involved with every update of an agent's contact address, violation of the HCD can be detected there. Conceivably, this check can take place probabilistically, to avoid the cost of checking the HCD each time a contact address is updated. A final approach is where the world owner simply uses agents to search a world for breach of policy at regular intervals.

[70] When zones trust each other and when jumping is not allowed, worlds may do without explicit enforcement of zone entrance policies and simply pass the RoomID to register in a hyperlink to each other.

.

Appendix 3
Jailer resource management

A requirement for a systems that runs arbitrary untrusted programs is *resource management*, in the sense of preventing the system from becoming starved of processor time, memory, or disk space by a malicious program that (directly or indirectly) uses too many of these resources. Current ‘‘off the shelf’’ operating systems like Linux are not particularly good at preventing attacks on system resources, particularly not at preventing against denial of service attacks, also called *resource exhaustion* attacks [71]. In Mansion, there is thus a need to cope with malicious programs that may mount such an attack. If possible, it is convenient to have the jailer This appendix describes some exploratory work on resource management in the jailer.

A jailer can do basic resource exhaustion prevention by enforcing global limits on system calls. Example limits are a maximum number of threads or processes that a prisoner may create, or that can run at the same time. The number of bytes that a prisoner may write to disk, or the amount of physical memory it may use are other examples. These limits are simple to enforce by keeping track of current use of a resource (e.g., memory) and checking system calls that modify the amount of resource usage against a limit. The jailer command line can take options to govern the above limits.

A more elusive form of resource usage which is harder to manage, is (system) time usage. A resource exhaustion attack may claim so many resources (e.g., CPU or system time) that other processes no longer get their share. Examples program behaviours that can trigger resource denial of service attacks are: invoking many system calls to *open, read* and *close* arbitrary files on disk (purging the disk cache continuously), creating large amounts of memory pages (particularly if this triggers swapping), or creating a large number of signal handler contexts by installing a signal handler and repeatedly (or recursively) sending a signal to invoke it. These attacks cannot always easily be prevented using limits on system calls or their arguments. For example, a maximum of *open* calls in total may cause a programs that legitimately reads many files to fail, while a maximum of open files per time unit is defeated by closing the file immediately after *open/read*, as described above. It is difficult to set limits right, and system time usage is difficult to precisely account for using system call monitoring to begin with.

Current UNIX systems such as Linux do not provide interfaces to allow users to control the (relative) resource use (e.g., in terms of system time) of their programs in a fine-grained

way. Schedulers on commodity operating systems attempt to schedule user processes fairly [111]. Standard schedulers are not well-equipped to deal with malicious user processes. Most schedulers are pre-emptive and primarily focus on dividing user mode time fairly between user processes. Malicious programs can use (excessive amounts of) system time or resources, for example by executing time-consuming or memory or CPU-intensive system calls, to attack a system.

Schedulers deal with managing system time consumption less effectively than with managing the user mode time consumed by processes. System calls execute in kernel mode, consuming system time. Some system calls cannot be pre-empted or stopped halfway; once executed, they must finish; others just take a lot of time to return, (e.g., disk access) or compete for scarce resources (e.g., when triggering a page fault causing a new memory page to be swapped in). Although many system calls can interleave, allowing other system calls or user processes to use the system while waiting for a resource to finish work, this is not always the case, and some system calls simply use more resources—take more system time—than others. There is a large difference between a *read* system call and a *getpid* system call.

System calls that result in creation of a thread of execution—for example, when creating a context for a registered signal handler—compete with other execution contexts (threads, processes). In general, system calls that require the kernel to do bookkeeping (e.g., process or memory management related calls) may take a relatively large amount of system time. Because of these intricacies, managing system time in the kernel properly and accounting for it when scheduling processes is difficult—furthermore, given the limited number of effective defenses, most operating systems seem not to expect that user processes mount denial of service attacks from the inside.

From a practical perspective, there are various ways in which malicious program can misuse system calls or kernel resources to deny other processes sufficient time to run. A straightforward example is a process which repeatedly executes a recursive search through the local file system, making use of time consuming system calls such as *open* or *read* to keep the disk busy. Particularly processes which also use disk may suffer from this attack. Another example is a recursive signal handler. Here, the kernel is kept busy creating signal handler contexts, which, when executed, create another one by sending a signal, and so on. From experience, by running such an attack on a stock Fedora distribution with a Linux 2.6 kernel, it is apparent that the kernel gives handling signals priority over scheduling regular threads or processes—a single program executing a recursive signal handler attack can make the system unresponsive. Similar is a *fork bomb*[71], which creates a large number of child processes using *fork* or, on Linux, *clone* system calls.

[71] In contrast to recursive signal handlers, fork bombs can be prevented in a straightforward way, by limiting the maximum number of threads or processes that a given process can create. On Linux, this can be done by controlling the *clone* call. A similar problem as outlined for *open/close* occurs though: does one limit the total number of clones in a jail, or the number of simultaneously running children? The approach outlined in this section focuses on rate-limiting an attack, by limiting the number of *clone* calls per time-unit. This can be combined with either one of the limit options. Combined with the second option, a trick to create a large number of processes which exit instantly will not be usable to mount a very successful denial of service attack.

A jailer that enforces resource control can help deal with above-mentioned attacks. By having the jailer limit the number of processes by controlling the *clone* system call, agents can be prevented from creating an excessively large number of child processes at a single time. Further, we assume that the user mode time is divided fairly using the operating system's (pre-emptive) scheduler, giving sets of processes in different jails a roughly equal share of the available user mode time. The jailer's primary task, then, is to manage system time. A difficulty for the jailer is its requirement that no changes may be made to the operating system.

An approach to manage system time in the Mansion jailer is explored in [54]. The approach consists of assigning an (estimated) *weight* in terms of system time to each system call, and enforcing system time use per jail by counting the actual use of system time per time slot. When a system call is invoked, the jailer predicts the total use of system time in the current time slot after the system call is made using the weight of this system call. If a user-defined amount of system time per time-slot for the jail has been exceeded, the system call is postponed until the next time slot. The procedure implemented in the prototype [54] is described in some detail below.

The jailer is instrumented so that it can *profile* jailed applications by measuring the time that system calls take to complete. This way, the jailer can assign a *weight* to each system call. System calls which are not executed by the jailed program during the profiling run, are assigned a (large) default weight by the jailer. Below is an example of an output file [54]:

```
SYS_write=8083
SYS_open=12121
SYS_fstat=200
SYS_mmap2=195
SYS_close=220
SYS_ioctl=126
SYS_read=148
SYS_munmap=237
SYS_lstat=8012
SYS_getxattr=20548
SYS_socket=218
SYS_fcntl=121
SYS_connect=412
SYS__llseek=233
SYS_readlink=9619
SYS_clock_gettime=26
SYS_getdents=447
SYS_mremap=47
SYS_brk=18
SYS_shmdt=56
```

A system call's weight is essentially the time a system call takes to complete under normal system load. It can be observed that there are some outliers in the list shown above; for example, write and open took more time than read, and getxattr also takes a significant amount of time. What is shown is the maximum time from a number of runs: a profiling run can take place several times, possibly with different programs; the maximum system time

observed in any of the runs for a given system call is kept in the output file. This approach was chosen to be on the safe side in estimating system time usage for system calls.

For system calls which are known to be usable for denial of service attack, the weight may be manually increased in the weight list, or using a second policy file or specific command line options (future work).

The jailer can run prisoners in *resource management mode*, which starts by reading in a weight list indicated on the command line. A command line argument is used to pass the main resource management parameter to the jailer: the *microseconds of system time* that all the processes in a jail may use per time slot. A time slot is currently a second.

While running, the jailer keeps track of the total system time used by a prisoner. Just as with profiling, the jailer does so by measuring the time between *syscall_pre* and a *syscall_post* events for all system calls. Each time a prisoner invokes a system call, in the *syscall_pre* handler, the jailer checks whether the maximum amount of used system time for the current time slot has exceeded the limit. If this is the case—or if the jailer estimates (by checking the weight list) that the current system call will cause it to exceeds the limit, if allowed—the jailer blocks the system call until the next time slot. The jailer does so by keeping the *ptrace* call that instructs the kernel to continue the system call pending until the next time slot. In all cases, the jailing policy is checked first.

The result of testing the approach is shown below The jailer runs a program, ‘‘ls –al /usr/*’’ in resource management mode, with the above profiling data read in from ˜/timingdata. ‘‘ls –al /usr/*’’ recursively lists the /usr/ directory on a standard Linux distribution. Below are shown the *system*, *user* and *real* time measured using the *time* command. In the first run, the prisoner is allowed to use 0.5 seconds of system time per second:

```
[cmd1] $ time bin/jailer --rmgt $PWD/timingdata -d ˜/mansion-jail/0 -p bin/jail-pol-
icy --rmgt-systime-us-per-sec 500000 ls -al /usr/*
 [... output omitted ...]
 real 0m5.689s
 user 0m0.883s
 sys 0m1.293s
```

The resulting execution time (wall clock time) of the program is around 5.69 seconds. Of this, 0.88 seconds are spent in user time, and 1.29 seconds is spent in system time, that is, in the kernel. The remaining time (*real* without *user* and *sys*) is used by other programs on the same machine.

Below the same program is run, now with a resource management allowing 0.25 seconds of system time per second:

```
[cmd2] $ time bin/jailer --rmgt $PWD/timingdata -d ˜/mansion-jail/0 -p bin/jail-pol-
icy --rmgt-systime-us-per-sec 250000 ls -al /usr/*
 [... output omitted ...]
 real 0m9.334s
 user 0m0.847s
 sys 0m1.437s
```

As can be observed, the total amount of system time and user time used by the prisoner is approximately the same as the previous run; this makes sense, as the number of system calls and the CPU time needed in user mode are identical for both runs. However, the real time, the wall clock time is 9.33 seconds, that is, the program's execution time is nearly doubled compared to the earlier run. This corresponds to expectations as the second program is allowed about half the system time per time slot of the first one; it thus takes longer to complete. In terms of resource usage, this means that the system call related resources that a prisoner can use are limited effectively. Repeated measurements using similar tests confirmed the above results (not shown). Note that the number of milliseconds per second is relatively high: the 0.5 seconds is selected to be around the system time needed for this program to execute without degradation, demonstrating that the total execution time doubles when constraining the system time per second to 0.25 seconds. Other programs may require significantly less system time per second, requiring a far smaller limit on the command line to have impact on the program's total execution time. Future work is needed to experiment with these setting for a number of benchmark programs, to establish a reasonable maximum to rate-limit denial of service attacks to a sufficient degree in practice.

Note also that there is a slight increase in the system time used in the second run, compared to the first. This difference may be attributable to the use of *alarm* system calls used by the jailer to implement time slot management and the extra time that system calls take while they are kept pending by the jailer.

In conclusion, the jailer can be used to impose a ceiling on prisoner's resource usage—say, 0.20 seconds per second maximum. Such a ceiling can prevent a single malicious agent from bringing the system to a halt, while probably not influencing most programs whose actual system time usage may stay well below this limit.

In practice, many issues may need to be resolved before systems can use the sketched approach reliably. For example, in Mansion, system time quotas per jail may need to be adjusted dynamically, to adapt to the resources available to a given jail to the number of other jails that are running at the same time. In other cases, some agents may be allowed to consume more resources than others, depending on for example a resource negotiation scheme [72].

The work described above provides a basis for future research. More in depth measurements and test cases are needed to explore how resource management behaves, particularly when other concurrent (jailed) processes are concurrently invoking system calls intensively—and how this influences the profiling results. A structured analysis of how specific denial of service attacks make use of particular system calls, and of how these may be alleviated and how the jailer functions under such attacks, would be very useful.

Finally note that tuning parameters (such as adjusting the weight of specific system calls depending on their potential for misuse) can turn out to be a challenge. For example, when assigning a large weight to the *kill* system call because it may be used to implement a recursive signal handler attack, other programs that make intensive benign use of this system call (such as a JVM) may suffer. Striking a balance may be difficult. Nevertheless, for defending

systems against kernel resource exhaustion attacks, using a jailer that is instrumented for resource management along the lines outlined above, seems a promising and appealing approach. The first tests, reported above, illustrate that the approach can work.

Appendix 4
Object replication

The MOS and object model described in this thesis (chapter 7) deals with nonreplicated objects, primarily focusing on security. It does not support replication directly. This appendix describes possible mechanisms and design options to include support for replication, which extend the Mansion object framework described in this thesis. It also describe a simple manual content replication scheme that uses a replicated file system. Because these approaches were not tested, we describe it as an appendix.

Motivation

The ideas sketched in this section are loosely inspired by the Globe distributed object system [18]. Globe is a distributed object system which is designed such that each object can implement its own *replication strategy.* In Globe, the replication strategy is implemented inside an object using an internal subobject called a replication subobject. In addition to a replication subobject, security and communication subobjects have also been designed [95, 16, 107], to deal with the various aspects related to object management and replication in large-scale distributed systems in an object-specific way. In contrast, Mansion takes a much simpler object model, where the MOS is tailored to Mansion's security model that uses ScID-based authentication, and which applies a standard security mechanism to all objects. The question is whether replication support can be embedded in the Mansion object system.

An example where replication is useful in Mansion, is for a popular room that is visited by many agents at the same time. Such a room can handle the load of many visiting agents by having a number of MMW processes in the zone, each running on a different machine. If the room contains an object with a large amount of data, for example, images, that agents search often, having a copy of this data on or nearby an agent's machine can prevent that the object that serves the content becomes a bottleneck. Providing each host or small set of hosts that serve agents with a MOS that contains a replica of the object is a solution to this problem[72].

[72] Ensuring that a directory with the data exists on each machine, which can be linked into the agent's jailing directory is a solution. However, this solution does not fit in the Mansion model where agents may currently only access objects. Future modifications to Mansion to support such a solution are discussed in Sec. 9).

Approach

An interesting starting point implementing replication in Mansion is that it provides a way to authenticate group members. Zones can be the basis for authenticated group communication, as zone members can authenticate each other as being member of the same zone. In Mansion, limiting replication to zones makes sense, as regular objects are accessible only in a zone. A more general object replication scheme could also make use of the zone concept. Using zones, different (sub)groups can be defined for object replication, similar to what was proposed in the context of Globe [60].

For example, a ‘‘core group’’ could use active replication to achieve consistency within a small set of trusted, hopefully highly available replicas, while one or more other, larger groups would obtain copies of (part of) the object’s state using, for example, master-slave replication [16] where a slave can connect to a master in the core group. A node or client outside the core group or the master-slave group would have to connect to a member of the slave group to invoke an object. Several policies, including differential policies (in which different methods have different replication policies) are conceivable.

From a security perspective, it would be straightforward to express a replication policy such as the above using ZoneIDs to specify groups.

Group-based policies can be relevant for several reasons. Among other things, differentiated replication policies could ensure availability (on a large scale) of nonsensitive and infrequently changing data, for example by means of master-slave replication. More sensitive data, on the other hand, could remain replicated and available only within a smaller, more trusted ‘‘core replica group.’’ Orthogonal to the data distribution policy of an object, an access control policy may also take zones into account, where invocations may be made from one zone and not from another.

Note that internally in a (Globe) distributed object, certain invocations that act upon sensitive data can automatically be propagated to the core replica set, whereas invocations working on nonsensitive data could be handled directly by a member of the master-slave replica set which has a copy of the nonsensitive state.

To discuss a concrete replication mechanism for the MOS, consider a simple, uniform replication scheme. If an invocation of a method causes changes to state (i.e., a *write* invocation), active replication can be used to distribute (replicate) this method invocation on all members of the group of object replicas. For now, assume that a reliable group communication mechanism can be established between all MOS processes within a zone. Reliable group communication mechanisms are implemented in, for example, the Isis toolkit [26]. More recently, several other reliable, ordered group communication mechanisms have been implemented, more or less resilient to failure (e.g., Paxos is an example of a group communication mechanism designed to be highly reliable even in view of arbitrary failure of participating nodes [57]).

Let us assume availability of a simple (but less reliable) sequencer service in each zone, to which all MOS processes can connect. The sequencer’s function is to issue sequence

numbers, using which the ordering of the ordering of messages can be reconstructed at the receiving side. Replicated invocations can be invoked in exactly the same order on all replicas of the object, ensuring consistency. Before sending a message to the group, a sequence number has to be requested. Next, the sender can multicast the message with this number included to all participants of the group, that is, to the set of MOSes containing replicas of the object.

For replicated objects, the set of of MOS processes which host a replica of a given object can be found by inspecting the location service. For the purpose of this section, the problem of leaving and joining groups as well as location service consistency are disregarded. Replication is not trivial, especially not in a large-scale system where nodes may fail arbitrarily. There is a reason it has not been implemented.

Assuming that there exists a reliable, ordered group communication infrastructure between MOS instances where every zone member can be connected to every other process in the zone using an authenticated ZAC channel, it becomes quite straightforward to design an active replication scheme for objects. Fig. 47 shows the idea.

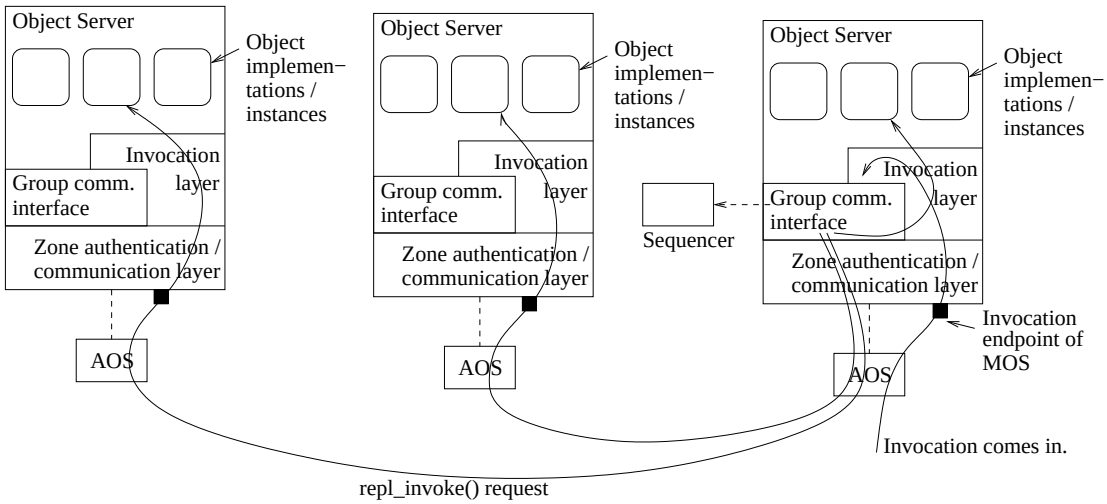

Fig. 47. An object replication scheme where the MOS manages active replication of method invocations for objects

In Fig. 47, a set of three MOS processes is shown. Each MOS contains a replica of the same object (the middle one). The figure does not distinguish the OM and OII layers explicitly; instead it focuses on the communication layer that underlies the RPC system; above that, the figure shows an Invocation layer which invokes the object; to simplify the figure we act as if OM and OII layers are integrated in a single layer, the **invocation layer**. In reality, invocation takes place through the OM and OII layers.

In Fig. 47, the MOS invocation forwarder service is extended with a *replicated_invoke* call, which accepts invocations from other processes in the same zone, which come in over

the communication layer. Replicated invocations carry a sequence number issued by the sequencer, and these can be executed in order by the invocation layer. Regular invocations take a different path. First, the MOS checks whether the method has to be replicated (see below). If so, a sequence number is requested from the sequencer, and the call is replicated (by placing the marshalled request in a *replicated_invoke* call) to all other MOS'es that host the object, within the same zone. Note that presumably, in a setup like this, MOS processes will have persistent connections open between them, over which replicated invocations on different objects can be sent.

In Fig. 47, an incoming invocation is shown to the right; the invocation layer notices that the incoming request needs to be replicated (how is described below). Next, the invocation layer passes the marshalled invocation to all other MOS processes which have a replica of the object by invoking a *replicated_invoke* call on these MOS'es. It does so through a group communication interface implemented on top of the ZAC layer. This group communication interface simply establishes point-to-point connections to other MOS'es which receive invocations through an RPC service. The invocation layer discards the original request, and waits for the *replicated_invoke* call to arrive through its communication interface. All the above is relatively straightforward, albeit clearly simplified from a proper reliable active replication system (i.e., it does not handle failures).

Replication bitmaps

An interesting question is how to indicate to the MOS what methods to replicate. The answer is simple: extend *MansionObject* with a method that allows an object owner to register a *replication bitmap* with the object. Similar to a role bitmap, the replication bitmap contains a bit per method, indicating whether it should be replicated or not. This allows an object designer to mark methods which do not have to be replicated as different from those which do. In general, only methods which cause a modification of an object's state need to be replicated. What methods modify state can typically be figured out straightforwardly from the object's IDL, knowing the object's semantics; in particular, *write* methods modify state, *read* methods do not. If necessary, an object's methods can be modified such that the semantics of every method is clear.
To support replication bitmaps, we can add the following methods to *MansionObject*:

```
set_replication_bitmap(bitmap)
must_replicate(method_id)
```

In the current MOS implementation, the *must_replicate* could be checked by the skeleton code in the OII after it checks the object's ACL. If the method is allowed and if it must be replicated, the OII can bounce the original marshalled invocation back to the OM layer using a runtime system method. (Note that the ScID (a 20-byte opaque field) and the *method_id* (an int32 following the ScID) can be checked before fully unmarshalling the

original invocation). Next, the OM layer can invoke the *replicated_invoke* call on the relevant peer MOS processes transparently using the group communication interface. The sequence number can be included in the *replicated_invoke* method. The OM and OII layer (including skeleton) must distinguish replicated from regular invocations to avoid replicating invocations recursively.

Manual replication

As a ‘‘quick fix,’’ objects can also be manually replicated. This is straightforward to implement. Identical content can be uploaded to manually created replicas of the object, each in a different MOS, which are registered under the same name. As an alternative, a shared distributed file system may be used underneath.

Mansion objects currently use the local file system to store (persistent) information for its objects. The current MOS implementation allows its owner to specify a directory (e.g., on NFS) which is used by the MOS and which contains directories accessible to objects. This shared directory can be shared by all objects in a zone. By allowing different object servers to access the same file system, it becomes possible to have each ‘‘replica’’ object access and serve the same state.

For a solution that operates at a larger scale (e.g., for wide-area replicated objects), similar solutions can be based on large-scale peer to peer data storage system like PAST (based on Pastry [101]). In general it is possible to reuse existing technology for building highly available file systems [102,112,125]. Alternatively, object replicas can interface with a (distributed) data base internally, or even with a dedicated file storage system such as, for example, designed for Grid systems [80]. Different instantiations of an object, each in a different MOS, can communicate with such an infrastructure internally while providing a distributed (replicated) object interface to the outside world. For a proper solution—similar to the proposal for object replication earlier in this section—the file system used should be aware of zones, or at least be able to authenticate and authorize all object replicas that attempt to access data.

If an object’s state does not change (often), manual replication can work fine. For as long as Mansion does not provide explicit support for object (state) or method replication, manual replication may provide an alternative.

.

Appendix 5
Location service resolver

This appendix describes how common entities are resolved on the Mansion location service using self-certifying names. The resolver embeds knowledge on naming conventions to locate entities in the location service.

In Mansion, ScIDs (usually, ZoneIDs) are used in all location-independent identifiers, such as object handles, agent identifiers, and the names of middleware processes and services which are part of the middleware. Some handles—like those of the services implementing the location service, MMW processes and object servers—are only used internally and not visible at the application level. At the application level, the most prominent handles are RoomIDs (for hyperlinks) and AgentIDs (for agents). These are visible to agents. ObjectIDs are registered in the RMO and visible to the MMW from there. Given some call of the agent, the MMW must figure out where, e.g., a MMW process is that belongs to a given zone that an agent follows a hyperlink to.

Starting from handles used at the application level internal handles can be constructed by means of a few simple naming conventions. For example given a ZoneID extracted from a RoomID, the self-certifying name of a middleware process in that zone can be constructed.

Naming conventions and resolving handles

All entities (agents, objects, and internal services) in Mansion have location-independent identifiers, called handles. The handle of an object is called an **object handle**. All handles have a standard layout:

```
<ScID>_<Type>_<Roomnumber>_<Objectnumber>
```

ScID is the base32-encoded self-certifying identifier of the process in which the object or service is located. Typically this is a *zone identifier (ZoneID)*, although it could be a *PeerID* if the process is not a member of a zone (Fig. 16).

Type is the name of the object's interface, or possibly of a service's interface. A **type** indicates a specific interface of a service or object as defined in the Mansion IDL (Sec. 7.3); Other examples are MMW or ATP types, which indicate the RPC interfaces that the MMW provides for setting up connections to agents and for agent transport, respectively. Types are

simple strings of up to 64 bytes.

Roomnumber is the number of the room relative to the zone. This number is relevant for hyperlinks, as a zone may have many rooms. For a service, the room number is set to 0, as services are not specific to a room.

Objectnumber is the number of the object relative to the room. If the identifier refers to a service and does not refer to a regular object, *Objectnumber* is set to 0. Roomnumber as well as objectnumber are encoded as a decimal number of up to 10 characters.

Any process in Mansion (including services and the MMW) with a remotely callable interface, may be named with a handle.

For illustration, example handles of services and objects in an actual Mansion world are:

```
6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_ALS_0_0
6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_Morgue_0_0
6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_RMO_0_0
6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_RMO_1_0
6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_ATP_0_0
6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_ZLS_0_0
2KXSFBINSIBAWHOCFKVR2QAHZF6DO4IG_RMO_0_0
2KXSFBINSIBAWHOCFKVR2QAHZF6DO4IG_MFC_0_0
2KXSFBINSIBAWHOCFKVR2QAHZF6DO4IG_MFC_0_1
2KXSFBINSIBAWHOCFKVR2QAHZF6DO4IG_ATP_0_0
2KXSFBINSIBAWHOCFKVR2QAHZF6DO4IG_ZLS_0_0
BR7WYHD7337FR7PZGTFZIIMFAU72N55H_Bootstrapper_0_0
BR7WYHD7337FR7PZGTFZIIMFAU72N55H_WLS_0_0
```

The names of the types of the handles above correspond to the Mansion services and components shown in from Fig. 6 and come from a location service cache of an existing world.

The list of handles illustrates important service and object types in Mansion:

- *ALS* the type of the agent location service;
- *ZLS* and *WLS* are types of location service components;
- *ATP* is the type for the externally reachable interface of the MMW process that can be used to ship agents to it using the Mansion agent transfer protocol;
- *Bootstrapper, WED, and morgue* are the types of services in the world entrance zone that play a role in bootstrapping a world, and in injecting and collecting agents;
- *MFC (MultiFileContainer)* is the type of a standard Mansion object used to store files.

For illustration, look at the list above. The structure of a world can be observed by just looking at the handles. Zone 6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB must be a world entrance zone, as an ALS and a Morgue have this ZoneID in their handle. Room handles that

end with RMO_0_0 are the first rooms created in a zone, and by convention,zone entrance rooms. 6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_RMO_0_0 must be a world entrance room. There is also a second world entrance room, room 1. There may be multiple world entrance zones per world for scalability reasons.

The second zone in this world 2KXSFBINSIBAWHOCFKVR2QAHZF6DO4IG contains a room with two MFC objects and no ALS, it is a regular zone.

The most important ScID is BR7WYHD7337FR7PZGTFZIIMFAU72N55H; this is the zone that contains the world's basement. It is recognisable because it contains the bootstrapper service. The ZoneID of the basement is used to authenticate world services such as the world location service and the bootstrapper service. Recall that handles are self-certifying, meaning that knowledge of—in this case—the WorldID is enough to authenticate the world services; trust extends from there (e.g., through the zone list which can be authenticated using the WorldID, and which can be used to find properties and ZoneIDs of zones in the world). Users authenticate the world location service and the bootstrapper services when they first initialise and configure a world. Once a middleware system is initialized with the WorldID (using the MLS and the WLS, Fig. 6), any component of the world can be found.

Examples

Using a process of incremental lookups, typically starting with looking up the ZLS of a zone, any object or service's handle can be resolved. As an example, assume a middleware process resolving the following object handle:

```
6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_MFC_1_0
```

Assuming a contact record of the WLS is known, the middleware locates the ZLS of the MFC object's zone, by constructing the handle 6T36FOGJAV2ZEDWDIFESUI7TQWYA6LB_ZLS_0_0 and resolving this handle in the WLS. Next, the ZLS is contacted to find the contact address (MCR) of the MFC object. An MCR is returned that contains the contact address of a MOS in which (one of the replica's of) the object resides; the index field in the MCR indicates the object within the MOS. If there are multiple MCRs, these are returned in round-robin order, unless the client resolver requests to obtain the MCRs of all replicas at once, e.g., to be able to select a (close-by) instance.

As indicated before, the location server is not trusted for security. It should be trusted to provide valid contact records to clients in general (if not, the world would stop to function as no content would be reachable from within the system), but a client can authenticate its peer end-to-end using the ScID from the peer's handle. However, the location service does basic checking: the location service that receives an update or a request to add or remove an MCR for a replica, first verifies that the registering process is in the zone corresponding to the handle and the MCR.

A final example, illustrating the process of finding a migration endpoint. A hyperlink looks like this:

```
6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_RMO_0_0
```

A hyperlink is a RoomID. The left-hand side of the RoomID is the room's 32-byte base32-encoded ZoneID. Using the RoomID, the middleware can straightforwardly construct a handle for an agent transfer protocol (ATP) endpoint in this room's zone by replacing the RMO_0_0 part of the object handle for ATP_0_0. The ATP endpoint is where a MMW processes waits for incoming agents.

```
6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_ATP_0_0
```

Using this name, the contact record for the ATP endpoints can be obtained from the zone location service of zone 6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB. which can be located by resolving the handle 6T36FOGJAV2ZEDWDIFESUI7TNQWYA6LB_ ZLS_0_0 in the world location service (WLS). There may also be multiple MMW processes in a zone ready to receive agents. These are registered under the same ATP handle.

Appendix 6
Optimising the clone protocol

The current protocol for cloning agents, described in Sec. 9.1.7, is inefficient, since it requires copying the parent's AC three times: for sending to the WED, when copying it *in* the WED, and for shipping it to the MMW that first starts the agent.

This appendix describes an optimisation. This optimisation also prevents overloading the WED with the task of physical cloning (copying) AC's, etc. Because the optimisation has not been implemented, we describe it in the appendix.

A MMW can clone the physical AC locally, without shipping it to the WED; instead, it can send the agent passport and ToC segments of the parent to the WED. (Only the MMW where an agent runs according to the ALS may do this). Next, the WED can (re)create and sign a new the agent passport and other initial segments for the new agent, including the initial ToC. For this, it uses the ToC checksums from the parent agent's ToC, to create a new ToC data structure which contains the ToC entries for the child's code and initial data segments, identical to the parent's code and (initial) data segments. To do this, the WED needs to know the ToC and initial segments of the original (parent) agent; it might have kept these from parent injection time, or the cloning MMW may send them to the WED including the signature by the WED over the parent's initial ToC. This signatures can be verified by the WED. Knowing this, the WED may register a new AgentID on the ALS and create the new initial segments including the TOC (which is identical to the parent's ToC except for the entry of updated segments, e.g., with the new AgentID) for the cloned agent, and send these to the cloning MMW process. The cloning MMW should ensure the new AC has the same segments as when the WED would have created it using the procedure outlined before.

The optimised cloning procedure is as secure as the non-optimised version, because the integrity of this new AC can and will be verified in the regular way, during the handoff protocol that takes place on the first physical migration of the new agent. A potential security risk exists: a cloning MMW can simulate the cloning procedure to obtain a new *AgentID*. However, the WED can check with the ALS that the cloning MMW indeed has the parent running on it. If so, the result of simulating the cloning procedure is no different than in the case where the cloning MMW would force a clone operation upon the parent agent against its will, and thus the risk is not exacerbated compared to the original clone protocol. (Mansion provides no protection of an agents against a malicious host that forces operations on it, or which substitutes an agent for a completely different piece of code at runtime; Mansion does not

provide a means to protect agents against the host at runtime except for the audit trail verification procedure when the agent returns).

With this restriction noted, optimising the clone procedure is possible and it has identical security properties to the non-optimised version. Doing this however requires careful programming, of both the MMW and the WED, where, among other things, a new ToC has to be created for the clone which matches the parent's ToC closely, but not precisely (i.e., the agent passport segment in the child's AC is different, leading to a different checksum than in the parent's ToC); also, the cloned AC's segment identifiers must match the ones in the ToC.

Appendix 7
Overhead of MMW to AOS RPC

In the AOS chapter, Sec. 4.7, throughput measurements and scalability of AOS were measured. This showed good scalability in that the total throughput remains constant independent of the number of concurrent sends or AC shipments over a single AOS-to-AOS base channel. In Sec. 8.3.2, ATP transfer cost were measured on a different set of machines.

Even though scalability is important for a kernel that is to be used by multiple processes concurrently, baseline performance in terms of latency and throughput are probably at least as important for most system designers. An important constraint with regard to AOS performance, is the fact that IPC (RPC) calls are made to invoke AOS operations. For heavyweight operations, such as an AC transfer, the added overhead is small compared to the overall cost of the (remote) operation and can be mostly neglected, but for tasks such as communication over an AOS channel, the additional RPC overhead may increase latency and decrease throughput.

To gain insight in this aspect, we measured the average round-trip time of a single invocation of a ‘‘ping’’ method on AOS kernel. The measurements include the time it takes the (multithreaded) SunRPC dispatcher over a TCP/IP substrate to handle the request, verify the cookie, and invoke the native ‘‘ping’’ method, which returns a 32 bit integer.

The average round-trip time of this RPC call is 129 µsec. For comparison, a simple *getpid* system call on the same machine takes 7 µsec on average. Roughly speaking, about 122 µsec is added when using an AOS primitive, compared to using an OS primitive (e.g., sockets) directly.

The RPC related overhead shows that AOS communication is not an optimal solution when low-latency, high-bandwidth communication is required; presumably, communication over AOS should be considered primarily in cases where there are limits on the number of usable TCP ports, for example, when a machine resides behind a firewall.

This measurement is placed in an appendix because it was made on the machine described in chapter 9, not on the same machine that the original AOS measurements in chapter 4 were made. The latter were no longer available at the time of making these measurements. The RPC measurements here are thus only intended to give a rough idea of RPC overhead using the AOS kernel.

.

Appendix 8
Using Mash—the Mansion shell

This appendix describes how a world is created and used from a user's and an administrator's perspective. It intends to illustrate how Mansion is used. The following sections present an administrator's shell and an interactive agent called *Mash*.

The Mansion shell

To facilitate administration of Mansion worlds, a program called the *Mansion shell (Mash)* has been developed. Mash is part of a set of utility programs that allow administrators to create new worlds and new zones. Command line tools exist to create and access RMOs and objects such as MFCs, to upload or download data to these objects. Mash makes use of these utility programs to create a unified environment, or *shell*, using which an administrator can conveniently use the relevant programs by means of familiar (UNIX-like) commands.

The Mansion shell always starts in the zone entrance room. In this room, an administrator can view the content of the room or a bound-to object, using a simple command like *ls* which, depending on context, invokes an RMO client or an MFC client program[73]. Mash is a stand-alone program; the programs that it spawns, like *rmocli*, connect to the MOS and authenticate using the zone member key that it (or, the room administrator) has access to. In a room, Mash can create a new room. A hyperlink is created to the new room, and by default a a back-link is created. The back-link, by convention, has the attribute *name*=.. The name of the new room, specified as an argument of the *newroom* command, is placed in the *name*= attribute of the hyperlink to the room.

To help a zone administrator navigate through its zone, Mash provides a simple shell-like interface. It has commands like *ls* (to view a room or an object's content), and *cd* (to enter another room, or to bind to an object in a room and enter it, as if it were a directory). All Mash commands use and act upon *name* attributes, but suppress the *name*= prefix. Attributes other than *name* are also suppressed. To view full (raw) attributes, a flag must be passed with a Mash command.

[73] Underlying commands, such as *rmocli* or *mfccli*, take a range of arguments, including file names of keys and certificates, and various options. Most of these arguments are hidden by Mash, which provides simple shorthand commands such as *ls* to list the content of a room, or the content of a multifile container.

By default, the Mansion environment appears as a directory structure with simple file names for rooms (hyperlinks) and objects. Mash provides an environment familiar to UNIX users. Mash is a tool for administrators. It is also turned into a mobile agent: Mash has become the *mobile agent shell.*

The mobile agent shell: Mash

By making some modifications to the Mansion shell, it becomes a mobile agent that allows for using Mansion interactively. To do so, modifications are needed:

1) The way in which the original Mash and its subprograms connect to RMOs and objects is modified. Instead of using a direct connection to the MOS (which administrative programs that have access to the zone key are allowed to do), a mobile agent must connect to and through the MMW using communication calls provided by the Mansion API. The required modification is straightforward, as the internal RPC calls to the RMO and other objects are identical; the only difference is in that connections and bindings by Mash must be made to the MMW RPC (forwarding) endpoint, through Mansion's binding mechanism, instead of to the MOS directly by means of a custom resolver and binding system.

2) Instead of using administrative programs to directly read object handles from RMOs and objects and resolving them, passing contact addresses on the client program's command line, the MMW now does this. Like any agent, Mash now uses calls relative to the Mansion context, using Mansion API methods. This is convenient.

3) Utility programs used by Mash, such as the *rmocli* program used to connect to the RMO, must similarly be modified. This mainly involves linking the programs with the Mansion API library and passing the MMW's communication endpoint (available to the Mansion shell as result of binding, see Sec. 2.3.10) to these subprograms instead of the endpoint normally passed on the client program's command line. The client program does it job, for example listing the content of an RMO or an object; the standard output of the client program is picked up by Mash (which spawns the client program and can capture standard in, out and error), which returns it over an interagent communication channel back to the agent's owner.

4) Instead of reading commands from standard input and writing results to standard output, mobile agent Mash listens for connections on an agent communication endpoint, receives commands from there, and returns them over the same channel to the user. A complication is that an agent owner's user interface is not an agent that runs in the world, so according to the Mansion model, the agent may not communicate with its owner. The solution is that the MMW where the agent resides accepts connections from

a user interface program which has access to the private key that corresponds to the AgentOwnerID (the hash of the agent owner's public key) of the agent. A better solution may be to create a ''proxy agent'' in the WEZ for the agent's owner, using which the agent owner can set up connections to the (child) agents it has in the world.

5) In contrast to the command line shell, the mobile agent shell is able to migrate (using *follow_hyperlink)* to another room. This requires modification of the Mash ''cd'' command to not simply bind to another object, but actually migrate. Migration is straightforward: since Mansion uses weak migration, Mash is shipped, killed at the source, and started from scratch at the destination. Then, Mash invokes the *accept* call to listen for a (re)connect from its owner, and waits to receive commands. The client program can retrieve the agent's current address from the ALS, and connect to the agent's new communication endpoint registered there after Mash migrated.

The above modifications allow Mash to run as an agent and execute commands for inspecting RMOs or objects, to download object content to the agent's jailing directory (resulting in it being stored in the agent's AC), etc. A command *exit* makes Mash exit, resulting in its AC being shipped back to the Morgue where it can be retrieved by the owner. From experience, Mash is a convenient way for agent owners to browse and explore a world. Fig. 48 shows a screenshot of running *Mash* as a mobile agent shell.

Fig. 48 gives a screenshot of a user injecting and steering Mash. This happens using a wrapper script (Mansion's ''main menu''). From this menu, the mobile agent shell is injected, after which a client program (*agentmonitor*) is started that connects to the agent. Agentmonitor connects to the agent, sends commands typed on the commandline verbatim to it, and waits for results, which are printed to standard out. If the connection to Mash closes (e.g., after sending a command causing the agent to migrate), agentmonitor receives a communication error. The user types a command to reconnect later.

The screenshot in Fig. 48 begins at the moment of injecting *Mash* by typing ''inject mash.'' The *inject* command invokes a program called *mansion_injector* with the pathname of the agent (Mash) as an argument. Some things can be observed from the inject commandline. First, the WED handle is known and filled in on the commandline by the main menu script. When the main menu script is started, an environment variable contains the WorldID or a default is read from a standard location in the user's home direction, ensuring that the script is set up in the context of a directory which contains all the information needed to do work in this world.[74] From this world's local configuration directory, the menu script obtains all the information needed to make user programs function, such as the keys required to authenticate to programs such as the world entrance daemon. The agent owner key is also found in the world's configuration directory. It is generated when Mansion is first used.

[74] Most information about a world is obtained when *bootstrapping* the world (see Sec. 3.3 and 3.3.4); this results in creation of a local directory containing the world's main configuration details.

As can be observed from Fig. 48, the keys and certificates that need to be loaded by the injector program, and are passed automatically to it: *agtk.pem*, in addition to contact information about the AOS kernel used to submit the agent to the world entrance daemon (WED)[75]. This information is filled in by the main script. *Inject* returns the global AgentID (*GAID*). After injection of *Mash*, control returns to the main menu.

The user starts up the program *agentmonitor* with the GAID as its argument. The *agentmonitor* is a modified MMW program that connects to the agent. Any command typed into the monitor (except for the *connect* command) is directly sent to the agent, after which the monitor simply waits for the response. Shown is execution of the *ls* command, which results in the Mash returning a listing of its current room's content. (Note that the monitor simply sends bytes as they are typed, as ASCII strings, to the agent; the monitor program is agnostic as to whether the agent is Mash or another agent).

Mash is started up in the world of Fig. 40, and it is asked to list the content of the room. The first room contains an object called *master-templates*, and a hyperlink to the next zone. When an agent is instructed to migrate through hyperlink 2 (*cd 2*), the agent disconnects, which is reported to the user. After a while, *connect* allows the user to reconnect to the agent, and the user can again type *ls* to have the agent report on the content of its current room using *ls*. The command *!ls* would report the content of the agent's current jailing directory (see below).

Mash is not an autonomous agent: it is controlled by a user. Using Mash, in contrast to the Web, a user can migrate through a (possibly world-wide) web of rooms, and access data here—for example, objects in the rooms, or data made accessible to the agent in the agent's jailing directory. The user has the full range of commands at the remote machine to his or her disposal, such as commands in /bin and /usr/bin. The remote system is protected by the jailer, which can control what information can be accessed by the agent, protects resources, and prevents the agent from connecting to arbitrary IP addresses. (This prevents, for example, a denial of service attack or other malicious attacks on systems reachable from the agent's current machine).

Using Mash, the full power of Mansion becomes noticeable to a user. Arbitrary code can be executed on arbitrary machines which are hooked into the network of rooms, so users can send agents to where the data is, and can have access to the raw system interface there (within the constraints of the jailer, Mash allows direct execution of UNIX commands available on the remote machine).

[75] As described earlier in this thesis, the mechanism to inject agents in the world is external to Mansion, as diverse mechanisms can be devised by the world entrance zone developer. For convenience, the prototype uses AOS for injection.

Fig. 48. Screenshot of using Mansion and the *mobile agent shell*. Shown are the general Mansion wrapper script, agent injection, and the agent monitor program. The ‘‘ls’’ command is used to list the contents of an agent’s room. Upon following a hyperlink (‘‘cd 2’’), the connection with the agent is broken and the user has to reconnect using the ‘‘connect’’ call.

Agents have access to the full power of the remote machine. This differs from the Web,

where a user can only retrieve Web pages from a remote location and display them locally. In a sense, *Mash* is a modern form of the UNIX guest account (which is removed on almost all machines today for security reasons, but was quite common until some 15 years ago).

Mash runs in a jailer, and provides access to UNIX commands as well as to rooms and content in them. Hyperlinks are presented by Mash as directories, and users can *cd* to one of them to make Mash follow the hyperlink. Similarly, Mash can ‘‘cd’’ into an *MFC* object, placing the user in the context of an object client program that provides a familiar ftp-like interface to view and download files from the object. ‘‘Cd-ing’’ out of the object puts the user back in RMO client mode to view the room and execute UNIX commands.

Experiences using Mash

The original stand-alone Mansion shell and *Mash* can be used to access rooms and objects, and also as a regular UNIX shell. It executes UNIX commands that are prefixed by a ‘‘!’’. For example, *!ls* does not invoke the Mash ‘‘ls’’ command for listing the content of a room, but it returns the content of the current working directory of the agent, typically the agent’s jailing directory.[76] It emulates familiar commands so that navigation through rooms becomes effortless, using commands like *cd*. In a sense, the agent now physically traverses a distributed ‘‘directory structure.’’

To the author of this thesis, the experience when using Mash is one of power and flexibility; having Mash execute commands on a remote machine directly, within a jail, while being able to flexibly migrate to other machines when needed (visiting other rooms and content there), feels impressive indeed.

Possible improvements

For the prototype, the user’s monitor program connects to Mash. For this, it contacts the ALS to find the agent’s current contact address. This works, but is somewhat suboptimal. Allowing any client program to query the ALS, irrespective of whether this client program is part of one of the world’s zones, is not preferable from a security perspective. The fact that user programs—even though these are authenticated as agent owner programs—removes some of the implicit trust-based security that comes with accepting connections from zones of a world only.

The solution is also suboptimal from the perspective of the agent’s owner. Having the user connect to the agent is suboptimal, since some time may be lost between when an agent is started, and when the user connects. When an agent owner instructs Mash to follow a hyperlink (*cd nextzone* or *cd 2*), Mash invokes the *follow_hyperlink* call, migrates, and after migration waits for a new command. The user has to guess when migration has completed (usually, the user just waits a few seconds), and then issues a ‘‘connect’’ command on the

[76] We have not implemented terminal emulation over the user-Mash communication channel, but this certainly would be feasible: in that case, arbitrary remote programs could be executed, resembling what can be done with an *ssh* terminal.

monitoring program. (Note that the monitoring program is initialised with the *AgentID* of the agent, so the user does not have to specify the AgentID again with the connect command). Monitoring multiple agents is possible by switching between connections.

The way in which an agent owner's programs connect to their agents also breaks with the Mansion model, as formally only agents within a world may connect to an agent. Furthermore, the way in which the MMW authenticates user programs using the agent's *AgentOwnerID* before allowing them to connect, does not work when the AgentOwnerIDs carried by agents indicate roles or payment schemes instead of agent owners, as described in Sec. 3.10. Also, the current implementation makes it impossible for users to remain anonymous with regards to the MMW systems (and thus the zones) that host their agents: if the agent owner connects, the MMW knows the agent owner's key and the IP address from which the agent owner makes contact; if this information is not sufficient to directly identify the user, it certainly allows linking different agents to a single user / IP address.

A solution could be to have the world entrance zone, upon accepting an agent, create a *proxy agent* for the agent's owner, to which the user's agent(s) can connect. This proxy agent would be registered in the ALS like any other agent, and its AgentID would be registered in the AC of the newly created agent such that the agent can ''phone home'' to its owner. (The Mansion API could be extended with an extra method for obtaining the owner proxy agent's AgentID). The owner can connect to the proxy, using its agent owner key to authenticate. The proxy can relay any command or reply messages from the user to the agent and vice versa. If agents connect back to the proxy agent right after migration, and if the proxy agent buffers commands from the user, this avoids the need for the user to reconnect to its agent and can provide migration transparency.

.

Appendix 9
Lightweight Mansion

This appendix describes a possible alternative to Mansion, where objects are replaced with directories mounted in the jailer, called *lightweight Mansion (LWM)*. Mash illustrates the potential of a future, simplified version of Mansion.

Lightweight Mansion (LWM) is simplified in the interface provided to agents: instead of binding to objects, agents access a directory structure available (mounted) in their jailing directory. Agents can then traverse a world by making system calls that take standard (UNIX) directory names as arguments, making use of existing UNIX tools (such as a standard shell). Extensions to the jailer are required to this purpose. The jailer can check whether directories or files that are being opened map to Mansion constructs (hyperlinks, objects) so that the jailer can translate these operations to Mansion operations transparently. Clearly, the standard semantics of directories in UNIX is changed in some cases.

An agent's jailing directory, can contain special (sub)directories, e.g., /hyperlinks/ to contain hyperlinks, and /shared/ or /objects/ to contain (object) data. There may also be an /agent/ directory which contains links to other agents (conceivably, *named pipes* could be used to communicate with other agents). Given knowledge of the Mansion paradigm, it is possible to see how such a directory structures can be mapped on (Mansion) semantics. Changing directory to a file in the /hyperlinks/ directory causes the agent to migrate to another room; this restarts the agent, possibly on a different machine, with another directory structure mounted into its jailing directory. Now the agent runs in the context of the target room, which is translated to a set of directories and files linked into the agent's jailing directory.

As described for Mash, a *chdir* to an object directory makes the object's data accessible using regular UNIX system calls. These can map on *MFC* calls internally (by having the jailer make Mansion API calls to bind to the object and invoke its methods), or if convenient from an implementation perspective, it is even possible to replace the Mansion object system such that an underlying (NFS) directory structure is accessed directly. This is transparent to the agent: in LWM, agents interact with the system using POSIX system calls, which are translated to relevant Mansion calls by the jailer, if required.

Attribute sets could be made available using a naming convention. For example, objects, hyperlinks and agents connection endpoints are directories or special files; by having regular files starting with the entity's file name with an extension, like *<entityfilename>.as*, agents

can be provided with files that contain the attribute set of the corresponding entity in simple ASCII attribute=value representation.

Mapping file system operations in a jail on remote objects in the jailer is an interesting thought. The jailer can be extended using a module to implement a user-level file system, internally using objects such as the *MFC* as a kind of distributed file system. Mapping Mansion completely on a distributed file system (assuming appropriate access and distribution control using zones are available) is also possible. LWM may use special directories for per-room bookkeeping—protected from agents, similar to how the RMO is used—while agents see only room-specific directories in their jailing directory. An agent sees directories and files in its private jailing directory, which, internally, map on Mansion constructs. Agents can use standard operating system calls and libraries to interact with Mansion, without having to be linked with the Mansion API or object interfaces. In LWM, all Mansion concepts are mapped to a file or directory in the agent's jailing directory. This provides agents with a consistent way to interact with their environment, and it provides full support for using legacy (UNIX) programs—like *grep* or *perl*—in Mansion without modification.

LWM can be constructed using a ''fat jailer,'' which modifies system calls to implement Mansion behaviour. The jailer can be instrumented to flexibly ''mount'' directories into the agent's jailing directory by constructing a jailing policy based on information in the RMO. Content may be provided by having *open* calls map on files in a *MFC* object, or they may be provided directly in an underlying (local) file system; the RMO could be extended with a way to store policies or local directory file names such that these can become available in an agent's jailing directory as read-only directories. The jailer transparently translates file system calls made by the agent into operations like binding or retrieving data from an object. Note that if data is accessible as files, so there is no need to download files to the jailing directory first.

The jailer is the core of LWM. This is not strange, as the jailer is also the core of Mash. Without the jailer, Mash would have far less potential. One of its strengths is that Mash provides a real remote (UNIX) shell powers to user, so that native applications and scripts can be executed securely in the context of a room, on a remote system. In LWM, agents could be implemented as scripts also, making existing scripting languages and their interpreters (or shells) available as tools to program mobile agents.

The above is just a thought experiment, and it raises many questions. For example, would global interagent communication still be possible, and if not, would it be missed? Would world structure and world boundaries still be intuitive? Mansion provides a clear model, whereas in LWM, all Mansion concepts are mapped on files and directories, possibly making the system's structure less obvious.

The LWM thought experiment illustrates how Mansion concepts can be applied in different ways. It is conceivable that an LWM implementation can be based on a simple (distributed) file system abstraction instead of a distributed object system (modulo zone-based distribution support), which may be convenient from a programmer's perspective. Having the jailer transparently map file access to objects is a useful thought for a future version of

Mansion where integration and reuse of legacy tools and libraries is straightforward.

Lightweight Mansion exploits a transparent mapping of Mansion concepts, such as hyperlink and local access to data, on file system operations by the jailer. Agents use a ‘‘private directory structure’’ as the interface to interact with the system, instead of following hyperlinks or binding to objects. This can allow unmodified legacy applications—for example, shell programs—to be used as mobile agents, while still providing all properties of Mansion and at the same time protecting the systems that host agents. So, Lightweight Mansion is Mansion without the intricacies needed to interface with distributed objects.

.

Acknowledgements

This thesis was in part made possible by financial support from Stichting NLnet, the Virtual Laboratory for e-Science (VL-e) project, and COMMIT/ .

.

Bibliography

References

1. "Trusted Computing Group (TCG) - Trusted Platform Module TPM 1.2 specification," *http://www.trustedcomputinggroup.org/.*
2. "Amazon Elastic Compute Cloud (EC2)," *http://aws.amazon.com/ec2/.*
3. "Subterfugue," *http://subterfugue.org.*
4. "Security Enhanced Linux," *http://www.nsa.gov/selinux/.*
5. "Strace program," *http://www.liacs.nl/ wichert/strace/.*
6. "VMware," *www.vmware.com.*
7. "FIPA Agent Communication Language (ACL) Specifications," *http://www.fipa.org/repository/aclspecs.html.*
8. A. Alexandrov, P. Kmiec, K. Schauser, "Consh: Confined Execution Environment for Internet Computations," *http://www.cs.ucsb.edu/ berto/papers/99-usenix-consh.ps* (1999).
9. R. Alfieri, R. Cecchini, V. Ciaschini, Ã. Frohner, A. Gianoli, K. LÅrentey, F. Spataro, I. Firenze, "An Authorization System for Virtual Organizations," *Proc. 1st European Across Gridss Conf., Santiago de Compostella* (2003).
10. D. P. Anderson, "Boinc: A system for public-resource computing and storage," *5th IEEE/ACM International Workshop on Grid Computing* (2004).
11. Ross Anderson, "Security Engineering, 2nd edition," *Wiley* (2008).
12. F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, B. Alberti, "The Internet Gopher Protocol (a distributed document search and retrieval protocol)," *RFC 1436* (1993).
13. Y. Artsy, R. Finkel, "Designing a process migration facliity: the Charlotte experience," *IEEE Computer* (1989).
14. G. Back and W. Hsieh, "Drawing the Red Line in Java," *Workshop on Hot Topics in Operating Systems (HotOS VII)* (1999).
15. G. Back, P. Tullmann, L. Stoller, W. C. Hsieh, J. Lepreau, "Java Operating Systems: Design and Implementation," *U. Utah Technical Report UUCS-98-015* (1998).
16. A. Bakker, "An object-based software distribution network," *PhD dissertation, Vrije Universiteit Amsterdam* (2002).
17. G. Ballintijn, "Locating Objects in a Wide-area System," *PhD Dissertation, Vrije Universiteit, Amsterdam* (2003).

18. A. Balogh, “New Object Server for Globe,” *Master’s Thesis, Vrije Universiteit, The Netherlands* (2003).
19. J. Baumann, F. Hohl, M. Strasser, K. Rothermel, “Mole - Concepts of a Mobile Agent System,” *Technical Report, Universit* (1997).
20. F. Bellifemine, A. Poggi, and G. Rimassa, “Developing Multi-Agent Systems with a FIPA-Compliant Agent Framework,” *Software -- Practice and Experience* **31**(2), pp. 103-128 (2001).
21. F. Bellifemine, A. Poggi, G. Rimassa, “Developing Multi-Agent Systems with a FIPA-Compliant Agent Framework,” *Software -- Practice and Experience* **31**(2), pp. 103-128 (2001).
22. S. G. Belmon, B. S. Yee, “Mobile Agents and Intellectual Property Protection,” *In K. Rothermel, F. Hohl (eds.), Proc. 2nd Int’l Workshop on Mobile Agents (MA)* (1998).
23. T. Berners-Lee,J. Hendler, O. Lassila, “The Semantic Web,” *Scientific American, May* (2001).
24. Tim Berners-Lee, “Original proposal for a global hypertext project at CERN (1989),” *Available at: http://www.w3.org/History/ 1989/proposal.html.*
25. W. Binder and V. Roth, “Secure mobile agent systems using Java: where are we heading?” *Proceedings of the 2002 ACM Symposium on Applied Computing* (2002).
26. K. P. Birman, “Reliable Distributed Systems - Technologies, Web Services, and Applications,” *Springer* (2005).
27. A. D. Birrell, B. J. Nelson, “Implementing remote procedure calls,” *ACM Trans. Computer Systems* (1984).
28. D. Chess, C. Harrison, A. Kershenbaum, “Mobile Agents: Are They a Good Idea?” *IBM Technical Report RC 19887* (1994).
29. F. Douglis, J. K. Ousterhout, “Transparent Process Migration: Design Alternatives and the Sprite Implementation,” *Software - practice and experience* (1991).
30. B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, R. Neugebauer., “Xen and the art of virtualization,” *Proc. ACM Symposium on Operating Systems Principles (SOSP)* (2003).
31. P. Efstathopoulos, M. Krohn, S. VanDeBogart, C. Frey, D. Ziegler, E. Kohler, D. Mazieres, F. Kaashoek, R. Morris, “Labels and Event Processes in the Asbestos Operating System,” *Proc. 20th Symposium on Operating Systems Principles (SOSP 2005), Brighton, United Kingdom* (2005).
32. R. Endsuleit, J. Calmet, “A Security Analysis on JADE(-S) V.3.2,” *Proceedings of NordSec, Estonia* (2005).
33. D. R. Engler, M. F. Kaashoek, J. O’Toole Jr., “Exokernel: an operating system architecture for application-specific resource management.” *Proc. Fifteenth ACM Symposium on Operating Systems Principles (SOSP)* (1995).
34. G. Fedak, C. Germain, V. Neri, “XtremWeb: A Generic Global Computing System,” *Proc. IEEE Int’l Symp. Cluster Computing and the Grid (CCGrid)* (2001).

35. D. Fensel, F. van Harmelen, I. Horrocks, D. L. McGuinness, “OIL: An ontology infrastructure for the semantic web,” *IEEE Intelligent Sytems* (2001).
36. N. Ferguson, B. Schneier, “Practical Cryptography,” *Wiley Publishing* (2003).
37. B. Ford and R. Cox, “Vx32: Lightweight user-level sandboxing on the x86,” *Usenix Annual Technical Conference (ATC)* (2008).
38. I. Foster, C. Kesselman, S. Tuecke, “The Anatomy of the Grid - enabling scalable virtual organizations,” *Int. J. Supercomputing Applications (15)* (2001).
39. T. Garfinkel, “Traps and Pitfalls: Practical Problems in System Call Interception Based Security Tools,” *Proc. Symposium on Network and Distributed System Security (NDSS)* (2003).
40. T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, D. Boneh, “Terra: a virtual machine-based platform for trusted computing,” *Symp. Operating Systems Principles (SOSP)* (2003).
41. T. Garfinkel, B. Pfaff, M. Rosenblum, “Ostia: A Delegating Architecture for Secure System Call Interposition,” *Proc. ISOC Network and Distributed System Security Symposium (NDSS)* (2004).
42. D. P. Ghormley, S. H. Rodrigues, D. Petrou, T. E. Anderson., “SLIC: An Extensibility System for Commodity Operating Systems.” *USENIX 1998 Annual Technical Conference* (1998).
43. Li Gong, *Inside Java 2 Platform Security: Architecture, API Design, and Implementation*, Addison-Wesley (1999).
44. T. Goodale, S. Jha, H. Kaiser, T. Kielmann, P. Kleijer, G. Von Laszewski, C. Lee, A. Merzky, H. Rajic, J. Shalf, “SAGA: A Simple API for Grid Applications. High-level application programming on the Grid,” *Computational Methods in Science and Technology* (2006).
45. R. S. Gray, D. Kotz, G. Cybenko, D. Rus, “D’Agents: Security in a Multiple-language, Mobile-agent System,” *Mobile Agents and Security* (1998).
46. V. Gunupudi, S. R. Tate, “SAgent: A Security Framework for JADE,” *Autonomous Agents and Multi-Agent Systems Conference (AAMAS), Japan* (2006).
47. Gutknecht, Olivier and Ferber, Jacques, “The MADKIT Agent Platform Architecture” in *Proceedings of the International Workshop on Infrastructure for Multi-Agent Systems*, pp. 48-55, Montreal, Canada (2000).
48. Fritz Hohl, “Time Limited Blackbox Security: Protecting Mobile Agents from Malicious Hosts,” *Mobile Agents and Security* (1998).
49. K. Jain, R. Sekar, “User-Level Infrastructure for System Call Interposition: A Platform for Intrusion detection and Confinement,” *ISOC Network and Distributed System Security Symposium (NDSS)* (2000).
50. D. Johansen, “Mobile Agents: Right Concept, Wrong Approach,” *5th IEEE Int’l Conf. Mobile Data Management (MDM)* (2004).
51. D. Johansen, R. van Renesse, and F. B. Schneider, “Operating Systems Support for Mobile Agents” in *Proceedings of the 5th Workshop on Hot Topics in Operating*

Systems, pp. 42-45, Orcas Island, WA (1995).

52. D. Johansen, K. J. Lauvset, R. van Renesse, F. B. Schneider, N. P. Sudmann, K. Jacobsen, "A tacoma retrospective," *Software - practice and experience* (2001).
53. P. H. Kamp, R. N. M. Watson., "Jails: Confining the omnipotent root," *Proc. 2nd Intl. SANE Conference* (2000).
54. J. Kamphorst, "Resource management in een Linux jailing systeem," *B.Sc. Thesis, University of Amsterdam* (2008).
55. N. Karnik and A. Tripathi, "Security in the Ajanta Mobile Agent System," *Software - Practice and Experience 31(4),* pp. 301-329 (2001).
56. C. Kaufman, R. Perlman, M. Speciner, *Network Security, 2nd ed.* Prentice Hall (2002).
57. L. Lamport, "The Part-Time Parliament," *ACM Transactions on Computer Systems 16, 2* (1998).
58. D. Lange and M. Othima, "Mobile Agents with Java: The Aglet API," *World Wide Web 1(3)* (1998).
59. E. Laure, C. Gr, S. Fisher, A. Frohner, P. Kunszt, A. Krenek, O. Mulmo, F. Pacini, F. Prelz, J. White, M. Barroso, P. Buncic, R. Byrom, L. Cornwall, M. Craig, A. Di Meglio, A. Djaoui, F. Giacomini, J. Hahkala, F. Hemmer, S. Hicks, A. Edlund, A. Maraschini, R. Middleton, M. Sgaravatto, M. Steenbakkers, J. Walk, A. Wilson, "Programming the Grid with gLite," *Computational Methods in Science and Technology* (2006).
60. J. Leiwo, C. Hanle, P. Homburg, C. Gamage, A. S. Tanenbaum, "A Security Design for a Wide-Area Distributed System," *ICISC* (1999).
61. N. Lhuillier, M. Tomaiuolo, G. Vitaglione, "Security in Multi-Agent Systems: JADE-S goes Distributed," *Exp. vol 3(3) - http://exp.telecomitalialab.com* (2003).
62. Z. Liang, V. N. Venkatakrishnan, R. Sekar, "Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs," *19th Annual Computer Security Applications Conference (ACSAC), Las Vegas, Nevada* (2003).
63. N. S. Malik, D. Ko, H. H. Cheng, "A secure migration process for mobile agents," *Software -- practice and experience (41),* pp. 87-101 (2010).
64. D. Mazières, M. F. Kaashoek, "Secure Applications Need Flexible Operating Systems," *Workshop on Hot Topics in Operating Systems (HotOS)* (1997).
65. D. Mazières, M. Kaminsky, M. F. Kaashoek, E. Witchel, "Separating Key Management From File System Security," *17th ACM Symposium on Operating Systems Principles* (1999).
66. S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," *15th Usenix Security symposium* (2006).
67. Sun Microsystems, "Solaris Containers: A Technical White Paper," *http://www.sun.com/software/whitepapers/ solaris10/grid_containers.pdf* (2004).
68. Sun microsystems, Inc., USA, *Java Remote Method Invocation Specification* (2004).
69. D. Milojicic, F. Douglis, R. Wheeler, eds., "Mobility: processes, computers and agents," *ACM Press* (1999).

70. D. Milojicic, M. Breugst, I. Busse, J. Campbell, S. Covaci, B. Friedman, K. Kosaka, D. Lange, K. Ono, M. Oshima, C. Tham, S. Virdhagriswaran, J. White, *MASIF The OMG Mobile Agent System Interoperability Facility* (1999).
71. Mitre, "Common Weakness Enumeration (CWE) 400: Resource Exhaustion," *http://cwe.mitre.org/data/definitions/400.html* (2013).
72. David Mobach, "Agent-Based Mediated Service Negotation," *PhD Dissertation, Vrije Universiteit, Amsterdam* (2007).
73. E. Mouw, G. J. van 'T Noordende, A. H. C. van Kampen, M. Santcroos, B. Louter and S. D. Olabarriaga, "Legal constraints on genetic data processing in European grids," *Healthgrid Conference, Amsterdam* (2012).
74. G. C. Necula, "Proof-Carrying Code," *24th Symposium on Principles of Programming Languages (POPL-97)* (1997).
75. R. van Nieuwpoort, T. Kielmann, H. E. Bal, "User-Friendly and Reliable Grid Computing Based on Imperfect Middleware," *Supercomputing (SC07) Reno, Nevada, USA* (2007).
76. G. J. van 't Noordende, "Comments on the definition of personal information and on the (re)use of personal information in anonymous or pseudonymised form in the proposed general data protection regulation." *A commentary* (2013).
77. G. J. van 't Noordende, B. J. Overeinder, R. J. Timmer, F. M. T. Brazier, A. S. Tanenbaum, "A Common Base for Building Secure Mobile Agent Middleware Systems," *Proc. 2nd Int'l Multiconference on Computer Science and Information Technology (IMCSIT), Wisla, Poland* (2007).
78. G. J. van 't Noordende, B. J. Overeinder, R. J. Timmer, F. M. T. Brazier, A. S. Tanenbaum, "Constructing Secure Mobile Agent Systems using the Agent Operating System," *Int. J. Intelligent Information and Database Systems (IJIIDS), Vol. 3, No. 4*, pp. 363-381 (2009).
79. G. J. van 't Noordende, F. M. T. Brazier, A. S. Tanenbaum, "Security in a Mobile Agent System" in *Proceedings of the First IEEE Symposium on Multi-Agent Security and Survivability*, pp. 35-45, Philadelphia, PA (2004).
80. G. J. van 't Noordende, S. D. Olabarriaga, M. R. Koot, C. Th.A. M. de Laat, "A Trusted Data Storage Infrastructure for Grid-based Medical Applications," *Proc. Eighth IEEE International Symposium on Cluster Computing and the Grid (CCGrid), Lyon, France* (2008).
81. G. van 't Noordende, "Agent Operating System (AOS) API, version 5rc3," *http://www.cs.vu.nl/ guido/aos/aosnotes/repository/aosnotes-v5rc3.ps* (2005).
82. G. van 't Noordende, A. Balogh, R. Hofman, F. M. T. Brazier. A. S. Tanenbaum, "A Secure Jailing System for Confining Untrusted Applications," *Proc. 2nd. Int'l Conf. on Security and Cryptography (SECRYPT), Barcelona Spain* (2007).
83. M. A. Oey, S. van Splunter, E. Ogston, M. Warnier, F. M. T. Brazier, "A Framework for Developing Agent-Based Distributed Applications," *Proc. 2010 IEEE/WIC/ACM Int'l Conf. on Intelligent Agent Technology (IAT-10), IEEE/WIC/ACM, Toronto, Canada*

(2010).

84. E. Ogston, “Agent Based Matchmaking and Clustering,” *PhD dissertation, Vrije Universiteit, Amsterdam* (2005).
85. S. D. Olabarriaga, P. T. de Boer, K. Maheshwari, A. Belloum, J. G. Snel, A. J. Nederveen, M. Bouwhuis, “Virtual Lab for fMRI: Bridging the Usability Gap,” *2nd IEEE Conference on e-Science and Grid Computing, Amsterdam* (2006).
86. Object Management Group (OMG), *CORBA/IIOP specifications.*
87. J. K. Ousterhout, J. Y. Levy, B. B. Welch., “The Safe-Tcl Security Model.” *Sun Microsystems Laboratories Technical Report TR-97-60* (1997).
88. M. X. Patel, V. Doku, L. Tennakoon, “Challenges in recruitment of research participants,” *Advances in Psychiatric Treatment (9),* pp. 229-238 (2003).
89. M. Payer and T. R. Gross, “Fine-grained user-space security through virtualization,” *ACM Virtual Execution Environments (VEE)* (2011).
90. M. Payer, T. Hartmann, T. R. Gross, “Safe Loading - a foundation for secure execution of untrusted programs,” *IEEE Symposium on Security and Privacy (Oakland)* (2012).
91. H. Peine, “Security Concepts and Implementation for the Ara Mobile Agent System,” *7th IEEE Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises* (1998).
92. H. Peine and T. Stolpmann, “The Architecture of the Ara Platform for Mobile Agents,” *Proc. First Int’l Workshop on Mobile Agents* (1997).
93. D. S. Peterson, M. Bishop, R. Pandey, “A Flexible Containment Mechanism for Executing Untrusted Code,” *Usenix Security Symposium* (2002).
94. A. Poggi, M. Tomaiuolo, G. Vitaglione, “Security and Trust in Agent-Oriented Middleware,” *R. Meersman and Z. Tari (Eds.), OTM Workshop, LNCS 2889,* pp. 989-1003 (2003).
95. B. C. Popescu, M. van Steen, A. S. Tanenbaum, “A Security Architecture for Object-Based Distributed Systems,” *Proc. 18th IEEE Annual Computer Security Applications Conference* (2002).
96. N. Provos, “Improving Host Security with System Call Policies,” *Proc. 12th USENIX Security Symposium* (2003).
97. RFC1832, “XDR: External Data Representation Standard,” *http://www.ietf.org/rfc/rfc1832.txt* (1995).
98. RFC2459, “Internet X.509 Public Key Infrastructure Certificate and CRL Profile,” *http://www.ietf.org/rfc/rfc2459.txt* (1999).
99. V. Roth, M. Jalali-Sohi, “Concepts and Architecture of a Security-centric Mobile Agent Server,” *Proc. 5th IEEE Int’l Symp. Autonomous Decentralized Systems (ISADS),* pp. 435-442 (2001).
100. V. Roth, U. Pinsdorf, J. Peters, “A Distributed Content-Based Search Engine Based on Mobile Code,” *Symposium on Applied Computing (SAC), Nicosia, Cyprus* (2004).

101.
A. Rowstron, P. Druschel, “Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems,” *Proc. 18th IFIP/ACM Int’l Conf. Distributed Systems Platforms (Middleware)* (2001).

102.
M. Satyanarayanan, “Scalable, secure, and highly available distributed file access,” *IEEE Computer* (2009).

103.
A. A. Schaffer, L. Aravind, T. L. Madden, S. Shavarin, J. L. Spouge, Y. I. Wolf, E. V. Koonin, S. F. Altschul, *Nucleic Acids Res. 29(14):2994-3005* (2001).

104.
B. Schneier, “Applied Cryptography,” *Wiley Publishers* (1996).

105.
J. Shapiro, “What is a capability, anyway?” *http://eros-os.org/essays/capintro.html.*

106.
T. Shinagawa, K. Kono, T. Masuda, “Flexible and Efficient Sandboxing Based on Fine-Grained Protection Domains,” *ISSS* (2002).

107.
M. van Steen, P. Homburg, A. S. Tanenbaum, “Globe: A Wide-Area Distributed System,” *IEEE Concurrency* (1999).

108.
W. R. Stevens, “UNIX Network programming,” *Prentice Hall* (1990).

109.
B. Stroustrup, “The C++ programming language, 3d ed.” *Addison-Wesley* (1997).

110.
N. Suri, J. M. Bradshaw, M. R. Breedy, P. T. Groth, G. A. Hill, R. Jeffers, T. S. Mitrovich, B. R. Pouliot, and D. S. Smith, “NOMADS: Toward a Strong and Safe Mobile Agent System” in *Proceedings of the Fourth International Conference on Autonomous Agents*, pp. 163-164, Barcelona, Spain (2000).

111.
A. S. Tanenbaum, “Modern Operating Systems, 2nd ed.” *Prentice-Hall* (2001).

112.
A. S. Tanenbaum, M. van Steen, “Distributed Systems - Principles and Paradigms 2nd ed.” *Pearson International Edition* (2007).

113.
A. S. Tanenbaum, R. van Renesse, H. Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, G. van Rossum, “Experiences with the Amoeba Distributed Operating System,” *Communications of the ACM vol. 33* (1990).

114.
D. Thain, T. Tannenbaum, M. Livny, “Distributed Computing in Practice: The Condor Experience,” *Concurrency and computation: practice and experience* (2005).

115.
A. R. Tripathi and T. Ahmed, Neeran M. Karnik, “Experiences and Future Challenges in Mobile Agent Programming,” *Microprocessor and Microsystems* **25**(2), pp. 121-129 (2001).

116.
Enforcing Integrity of Agent Migration Paths by Distribution of Trust, “M. Warnier, M.A. Oey, R.J. Timmer, B.J. Overeinder, F.M.T. Brazier,” *Int. J. of Intelligent Information and Database Systems (IJIIDS)* (2009).

117.
D. Tsafrir, T. Hertz, D. Wagner, D. Da Silva, “Portably Solving File TOCTTOU Races with Hardness Amplification,” *Usenix FAST* (2008).

118.
H. Vijayakumar, J. Schiffman, T. Jaeger, “Process Firewalls: Protecting processes during resource access,” *Eurosys symposium, Prague* (2013).

119.
X. Vila, A. Schuster, A. Riera, “Security for a Multi-Agent System based on Jade,” *J. Computers and Security (26), Elsevier,* pp. 391-400 (2007).

120.
W3C, “Resource Description Framework (RDF) Schema Specification 1.0,” *W3C Candidate recommendation, http://www.w3.org/TR/2000/CR-rdf-schema-20000327/* (2000).

121.
R. Wahbe, S. Lucco, T. E. Anderson, S. L. Graham, “Efficient software-based fault isolation,” *ACM Symposium on Operating System Principles (SOSP)* (1993).

122.
K. Walker, D. Sterne, L. Badger, M. Petkac, D. Shermann, K. Oostendorp, “Confining root programs with domain and type enforcement (DTE).” *Proc. 6th USENIX Security Symposium* (1996).

123.
D. S. Wallach, D. Balfanz, D. Dean, E. W. Felten, “Extensible Security Architectures for Java,” *16th ACM Symposium on Operating Systems Principles* (1997).

124.
X. Wang, Y. Yin, H. Yu, “Finding Collisions in the Full SHA-1,” *Proc. Crypto* (2005).

125.
S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, C. Maltzahn, “Ceph: a scalable, high-performance distributed file system,” *7th USENIX Symp. on Operating Systems Design and Implementation (OSDI), Seattle, WA* (2006).

126.
J. E. White, “Telescript Technology: Mobile Agents,” *White paper, General Magic* (1996).

127.
N. J. E. Wijngaards, B. J. Overeinder, M. van Steen, F. M. T. Brazier, “Supporting

Internet-Scale Multi-Agent Systems," *Data and Knowledge Engineering 41(2-3)* (2002).

128. I. Zupancic, "Redisigning a Linux jailing system," *M.Sc. Thesis, University of Amsterdam* (2009).

.

.

Index